A FAST ALGORITHM FOR

MINING THE LONGEST FREQUENT ITEMSET

FU QIAN

NATIONAL UNIVERSITY OF SINGAPORE

2004

A FAST ALGORITHM FOR

MINING THE LONGEST FREQUENT ITEMSET

FU QIAN

(B.Sc., Peking University)

A THESIS SUBMITTED

FOR THE DEGREE OF MASTER OF SCIENCE

DEPARTMENT OF COMPUTER SCIENCE

NATIONAL UNIVERSITY OF SINGAPORE

2004

Acknowledgement

I am most grateful to my supervisor, Professor Sam Yuan Sung, for guiding me through the master studies. He has consistently provided me with valuable ideas and suggestions during my research and been very considerate all the time. His continued support and deep involvement have been invaluable during the preparation of this thesis.

I would like to thank my best friend, Chen Chao, who is a NUS master and now a PhD candidate in RPI, for his very valuable comments and suggestions. Many in-depth discussions with him have been of great help in my research. Also thank him for his encouragement when I met difficulties.

I would like to thank Dr. Johannes Gehrke for kindly providing me with the MAFIA source code.

I would like to thank all the friends I met in Singapore, Chen Zhiwei, Chen Xi, Jiang Tao, Wu Xiuchao, Yuan Ling, Guo Yuzhi, Qian Zhijiang, Chen Wei, Lu Yong, Pan Xiaoyong...The enjoyment we shared together made my life in NUS more colorful.

I would like to thank my parents, Fu Yunsheng and Qian Yufen, for their unconditional support over the years. I would like to thank my father-in-law, Wang Jian, and my mother-in-law, Song Junyi, for encouraging me to pursue my master degree. I would also like to thank my elder brother and his wife, Fu Peng and Ma Qianhui, for taking care of our parents. Finally, I thank my wife, Wang Jing, for sharing my feelings during the time either I was happy or I was frustrated.

Table of Contents

Summary

Mining frequent itemsets in databases has been popularly studied in data mining research since many data mining problems require this step, such as the discovery of association rules, data correlations, sequential or multi-dimensional patterns. Most existing work focuses on mining frequent itemsets (**FI**), frequent closed itemsets (**FCI**) or maximal frequent itemsets (**MFI**). As the database becomes huge and the transactions in the database become very large, it becomes highly time-consuming to mine even the maximal frequent itemsets.

In this paper, we define a new problem, *finding only the longest frequent itemset from a transaction database*, and present a novel algorithm, called LFIMiner (Longest Frequent Itemset Miner), to solve this problem. Longest frequent itemset (**LFI**) can be quickly identified in even very large databases, and we find there are some real world cases where there is a need for finding the longest frequent itemset.

With the database represented by the compact FP-tree (Frequent Pattern tree) structure, LFIMiner generates the longest frequent itemset by a pattern fragment growth method to avoid the costly candidate set generation. In addition, a number of effective techniques are employed in our algorithm to achieve better performance. Two pruning methods, respectively called Conditional Pattern Base Pruning (CPP) and Frequent Item Pruning (FIP), reduce the size of the FP-tree by pruning some noncontributing conditional transactions. Furthermore, the Dynamic Reordering (DR) technique helps reduce the size of the FP-tree by keeping more frequent items closer to the root to enable

more sharing of paths.

We also performed a thorough experimental analysis on the LFIMiner algorithm. First we evaluated the performance gains of each optimization component. It showed that each of the components improved performance, and the best results were achieved by combining them together. Then we compared our algorithm against modified variants of the MAFIA and FPMAX algorithms, which were originally designed for mining maximal frequent itemsets. The experimental results on some widely used benchmark datasets indicate that our algorithm is highly efficient for mining the longest frequent itemset. Further, our algorithm also scales well with database size.

An application of **LFI** is to use **LFI** for transaction clustering. A frequent itemset represents something common to many transactions in a database. **LFI** is the kind of frequent itemsets with maximum length, and intuitively transactions sharing more items have a larger likelihood of belonging to the same cluster. Therefore, it is reasonable to use **LFI** for transaction clustering. We propose a clustering approach which is based on **LFI** and experiments on some real datasets show that this approach achieved similar or even better results than existing algorithms, in terms of class purity.

List of Figures

List of Tables

Chapter 1

Introduction

1.1 What is Data Mining?

Data mining, also popularly called as knowledge discovery in databases (KDD), is a multidisciplinary field, referring to areas including database technology, artificial intelligence, machine learning, neural networks, statistics, pattern recognition, knowledge-based systems, knowledge acquisition, information retrieval, high-performance computing, and data visualization [HK01]. As indicated by the literals, data mining is a process of discovering or *mining* interesting knowledge from large amounts of *data*. It develops data analysis tools to help people detect, understand and further utilize the valuable knowledge (*categories*, *patterns*, *concepts*, *relationships*, *trends*, etc.) embedded in the data "sea".

As database and information technology has been evolving and maturing since the 1960s, also the steady progress of computer hardware technology has made large supplies of powerful computers and storage media available, automated data collection equipment has led to tremendous amount of data collected and stored in large and numerous databases. However, people always feel perplexed in the face of such a large amount of raw data because it has so far exceeded our human ability of comprehension that we don't

know what information inside is useful. If we cannot make use of it, collecting and storing data in databases regrettably falls into lost labor. How can we transform “obscure” raw data into “explicit” information, which can help us make decisions on our next move? Data mining techniques emerge timely to change this “data rich but information poor” situation. They perform data analysis and uncover possible important patterns, contributing immensely to business and scientific research.

Although it is a young field, data mining develops very fast and becomes an important technology both for business strategies making and scientific research conducting. Much work has been done in order to perform data mining in large databases in an efficient and effective way. The major issues involved include mining methodology, user interaction, performance and scalability evaluation, the processing of diverse data types mined, and so on [HK01].

1.2 What Kinds of Patterns Can Be Mined?

There are various types of data stores on which data mining can be performed, such as relational databases, data warehouses, transactional databases, spatial databases, multimedia databases and the World Wide Web. Also there are various kinds of data patterns that can be mined. In this section, we examine some major data mining technologies and the kinds of patterns that can be discovered by them.

1.2.1 Data Characterization and Discrimination

It is clear and useful to describe individual groups of data in summarized and precise terms. For example, in a bar, customers can be sorted into two groups including

liquorDrinkers and *softDrinkers*. It is very clear for us to know that the first group of people drink beer, brandy while the second group choose syrup, soda water.

Data characterization is a process to summarize the general characteristics of a collection of data. The data can be retrieved through database queries. For example, after summarizing the characteristics of customers who drink beer or brandy in the bar, we could find some generalized information, such as they are male, between 30 and 50 years old, and have a good job.

Data discrimination is a comparison of the general characteristics of one class of data with the general characteristics of other contrasting data classes. Like data characterization, data of a specific class can be collected by a corresponding database query. For example, after comparing the two groups of customers *liquorDrinkers* and *softDrinkers* in the bar, we could get such a kind of generalized comparative profile as 80% of customers who drink beer or brandy are male, between 30 and 50 years old, and employed, whereas 70% of customers who drink syrup or soda water in the bar are young and students.

1.2.2 Association Rules Mining

The problem of mining association rules is to find interesting relationships among a given dataset. An example of such a rule might be that 80% of customers who buy pencils also buy erasers. Finding all such customer behaviors is valuable for retailers to develop their marketing strategies, for instance, placing pencils and erasers closely may encourage the sale of both items.

We give the formal statement of association rules mining as follows: Let $I = \{I_1, I_2, \ldots, I_N\}$ be a set of N distinct items. Let D be a set of database transactions where each transaction T is a set of items such that $T \subseteq I$. A transaction T is said to contain X, which is also a set of items, if $X \subseteq T$. An association rule is an implication of the form $A \Rightarrow B$, where $A \subset I$, $B \subset I$, and $A \cap B = \Phi$. The rule $A \Rightarrow B$ holds in database D with *support s* if s% of transactions in D contain both A and B, i.e. $A \cup B$. The rule $A \Rightarrow B$ has *confidence c* if c% of transactions in D that contain A also contain B. An example rule is buys (pencil) $\Rightarrow$ buys (eraser) {support = 20%, confidence = 80%}. It indicates that 20% of customers purchase both pencil and eraser, and 80% of that who have bought pencil also buy eraser.

Notice that we specify two interestingness measures, support and confidence, to estimate whether the rules found are interesting. In general, each measure is associated with a *threshold* that can be controlled by the user. Rules that do not meet the threshold are thought as uninteresting. The problem of mining association rules is to generate all the rules with support bigger than the user-specified minimum support threshold and confidence bigger than the user-specified minimum confidence threshold.

In general, association rule mining is a two-step process:

- Find all frequent itemsets according to a user-specified minimum support threshold: An itemset is a set of items, and an itemset is called a frequent itemset if the number of transactions that contain the itemset is as least as the minimum support threshold.
- Generate interesting association rules from the frequent itemsets.

The first step determines the overall performance of association rule mining because it is

most time-consuming. Many efforts have been conducted to look for efficient methods to find the frequent itemsets. This thesis dwells on this problem as well. The second step is an easier step, for detailed information, please refer to [AIS93], which describes how to generate association rules from the frequent itemsets.

1.2.3 Classification and Prediction

Classification is the process to find the functions that can distinguish data classes, and further use these functions to predict the classes of objects whose class labels are unknown. From its definition, classification is also a two-step process. In the first step, a sample database is given, known as training data, and each tuple in the database has a class label indicating the predefined class it belongs to. By analyzing these training samples, a function or model is derived to distinguish different classes. This step is also called as *supervised learning* because the model knows (is supervised) the class of each sample tuple and what the model needs to analyze is how one tuple belongs to a known class.

In the second step, the model is used for classification. But first, its predictive accuracy needs to be evaluated. A simple evaluating technique is to use a set of test samples with known class labels. The accuracy of a model is reflected by the percentage of test samples which are correctly classified by the model. Note the test set should be different from the training samples, which lack universality because the model is derived from them. If the accuracy is considered acceptable, the model can be used to classify future data tuples for which the class label is unknown. For example, the bank can use the information of existing customers to predict the credit rating of future customers.

There are many techniques for classification, including decision tree induction, naïve Bayesian classification, Bayesian belief networks, backpropagation, association rule mining, nearest neighbor classifiers and case-based reasoning classifiers.

Classification is used for predicting the class label of data objects, whereas *prediction* can be viewed as the process of predicting numerical values of some attributes, such as the salary of fresh graduates from NUS, rather than class labels. Like classification, prediction builds models based on historical data for predicting future behaviors. However, to predict continuous values, different techniques need to be applied, such as linear regression, multiple regression and nonlinear regression.

Classification and prediction are widely used in credit approval, medical diagnosis, performance prediction, selective marketing, and so on.

1.2.4 Clustering

Data clustering is a popular topic in data mining field for its capability of automatically partitioning database into clusters such that objects within the same cluster are as similar as possible, whereas objects of different clusters are as dissimilar as possible. These discovered clusters are used to explain the characteristics of the data distribution.

Clustering has been widely used in many applications, such as pattern recognition, data analysis, image processing, and web document classification. In business, marketers can seek help from clustering to identify distinct customer groups based on different purchasing patterns and accordingly arrange different selling schemes. In biology,

clustering can be used to categorize genes with similar functionalities, or derive taxonomies for different species. It can also be used in the Web to help classify documents for discovering significant information.

Unlike classification, clustering is considered as a process of *unsupervised learning* because there are no class labels it relies on. Its mission is to "cluster" similar unclassified objects together and segregate dissimilar objects from each other. An effective clustering can produce high intra-cluster similarity and at the same time high inter-cluster dissimilarity. We can deem each cluster as a class, and any class contains similar objects and is different from other classes (clusters). There exist some major clustering methods, *partitioning* methods, such as *k*-means, *k*-medoids, to allocate objects into *k* partitions of the data and each partition represents a cluster; *hierarchical* methods, further to be classified into *agglomerative* and *divisive* categories, to decompose the data hierarchically in a bottom-up or top-down manner; *density-based* methods to group objects based on density rather on the distance between objects, and so on.

Clustering is a challenging topic in data mining field and this thesis will also address a novel clustering method on categorical data.

1.2.5 Outlier Analysis

Outliers are a set of data objects that behave considerably dissimilarly from the rest of the data. For example, people whose lifespan is over 100 years are thought as *outliers* in the humankind world. In many data mining field, such as clustering, outliers can cause negative influence on the results and algorithms always try to minimize the impact of

them. However, outliers themselves may be of particular interest, for example, studying the living ways of those long-life people may provide beneficial suggestions to our current life. Also in the fields such as fraud detection, outliers may indicate fraud behaviors and outlier detection and analysis could help maintain a more regulated environment.

1.3 Research Contribution

This thesis will address a new problem that has not been explored before, namely *finding the longest frequent itemset from a transaction database*. Although mining maximal frequent itemsets (**MFI)** is much faster than mining frequent closed itemsets (**FCI)** or frequent itemsets (**FI)**, as the database becomes huge and the itemsets in the database become very long, it still becomes highly time-consuming to mine even **MFI**. In contrast, longest frequent itemsets (**LFI)** can be quickly identified even in very large databases because the number of longest frequent itemsets is usually very small, and may even be 1. In some real world applications, there is a need to find **LFI**. Consider a case where a travel company is to propose a new tour package to some candidate places. The company conducts a survey of its customers to find their preferences among these places, i.e. which places they want to visit. Suppose that the company wants the package to satisfy the following requirements: a) the number of customers taking this tour should be no less than a certain number, for instance, 20 (*quantity* requirement), and b) the profit per customer is maximized (*quality* requirement). Here, we assume the profit per customer is proportional to the number of places in the package. In addition, a customer is assumed to be cost conscious, i.e., he/she will not pay for the package if the package

contains the places he/she does not want to visit. This problem can be solved by finding **LFI** from the survey data having support $\geqq$ 20. Places in a longest frequent itemset constitute a desired package. There exist many analogous problems: for example, an insurance company wants to design an insurance package to attract a sufficient number of customers and maximize the number of insured subjects, or a supermarket wants to design a binding sales plan to maximize the number of items purchased together by a sufficient number of customers. This kind of problem can be well solved by finding **LFI**.

Another application of **LFI** is to use **LFI** for transaction clustering. A frequent itemset represents something common to many transactions in a database. Therefore, it is a natural way to use frequent itemsets for clustering. [BEX02] [FWE03] apply frequent itemsets into document clustering. In their strategies, documents covering the same frequent itemset are put into the same cluster. Note that **LFI** is the kind of frequent itemsets with maximum length, and intuitively transactions sharing more items have a larger likelihood of belonging to the same cluster. Therefore, it is reasonable to use **LFI** for transaction clustering.

An approach based on **LFI** for clustering transactions is briefly described in the following (with algorithm description and experimental results presented in Chapter 5). Our approach is divided into a partition phase and a refinement phase. In partition phase, transactions are stratified into clusters using **LFI** in a recursive procedure. In refinement phase, some adjustments are made. For example, given a cluster formed by a longest frequent itemset, the transactions containing a majority of items of the longest itemset may be moved into this cluster. Experiments on real datasets show that this approach

achieved similar or even better results than existing algorithms [GRS99] [ST02] [WXL99] [XD01] [YCC02], in terms of class purity.

In this thesis, we propose a solution to the problem of *finding the longest frequent itemset from a transaction database*. We have noticed that the FP-tree structure is useful for storing a database in compressed format, and as a depth-first algorithm, FP-growth has advantages in mining long frequent itemsets, since longer frequent itemsets may be detected earlier than some shorter ones. For our purpose of finding the longest frequent itemset, those shorter ones are not of interest and need not be generated. Due to the above benefits, we construct our algorithms based on FP-tree and FP-growth. One of our algorithms is LFIMiner for mining only *one* longest frequent itemset, and the other is LFIMiner_ALL for mining *all* longest frequent itemsets (**LFI**). In addition, some modifications are made to the original FP-growth algorithm and several optimizations are used to improve performance.

The principal weakness of the FP-growth algorithm is that it requires that the FP-trees fit in the main memory. With the size of computers' main memories growing continuously, many moderate to large databases can have their FP-tree structures completely kept in memory. In addition, mining **LFI** will not construct so many conditional FP-trees as mining **FI** because of the small number of longest frequent itemsets. Furthermore, due to effective pruning, our algorithm results in much smaller FP-trees compared with the algorithm without pruning. For example, for the *Chess* dataset, which contains 3,196 transactions with 37 items in each transaction, when the minimum support is 1%, the total number of nodes in the FP-trees without pruning is

22,209,818, whereas the number of nodes in the FP-trees with pruning is 575,394, which represents a reduction ratio of 38.60. As the support decreases, the ratio often increases further. All these factors make LFIMiner applicable to larger databases than the FP-growth algorithm. Therefore in our discussion, we assume that the FP-trees fit in the main memory.

With some widely used benchmark datasets, a thorough experimental analysis on our algorithm is performed. We compare our algorithm against modified variants of the MAFIA and FPMAX algorithms, which were originally designed for mining maximal frequent itemsets. We find LFIMiner is a highly efficient algorithm for finding the longest frequent itemset, and also it exhibits roughly linear scaleup with database size.

1.4 Thesis Organization

The rest of this thesis is organized as follows: Chapter 2 introduces the preliminary concepts about frequent itemset mining problem and reviews some related work. First we give the formal definitions of the terms and the problems addressed in this thesis. Then we will describe the conceptual framework of the item subset lattice on which frequent itemset mining algorithms base. After that, we will review some related research on frequent itemset mining. Some well-known algorithms, such as Apriori, MaxMiner, MAFIA and FPMAX, will be explored.

Chapter 3 first gives a brief introduction to the FP-tree structure and the FP-growth algorithm, and then describes our variant of the FPMAX algorithm. After describing the optimizations for reducing the search space, the LFIMiner and LFIMiner_ALL

algorithms are presented at the end of this chapter.

The experimental results are shown in Chapter 4, including an extensive study of the components of the LFIMiner algorithm and a comparison with the variants of MAFIA and FPMAX on real datasets. The results consistently prove our claim that LFIMiner is an efficient algorithm to find the longest frequent itemset.

In Chapter 5, we describe our approach which uses **LFI** for transaction clustering. We test our approach on some real datasets and the results show that our approach achieves similar or even better results than some existing algorithms, in terms of class purity.

We conclude in Chapter 6 with a discussion of future work.

Finally, the variants of the MAFIA and FPMAX algorithms are presented in the Appendix.

Chapter 2

Preliminaries and Related Work

2.1 Problem Definition

First, we give the formal definitions of terms and problems addressed in this thesis. For ease of exposition, we use the following notations throughout this thesis:

I	A set of items
D	A transaction database
ξ	A user-specified minimum support either in absolute or percentage number
FI	Set of frequent itemsets
FCI	Set of frequent closed itemsets
MFI	Set of maximal frequent itemsets
LFI	Set of longest frequent itemsets

Table 2.1: Notations

Let $I = \{I_1, I_2, \ldots, I_N\}$ be a set of N distinct items in a transaction database D. Each transaction T in D is a set of distinct items such that $T \subseteq I$. We call $X \subseteq I$ an itemset. An itemset with k items is called a k-itemset. The *support* of X *supp*(X) is the number of transactions containing X. Definitions of the terms and problems are presented as follows:

Term Definition 2.1 (Frequent Itemset): Let D be a transaction database over a set of distinct items I. Given a user-specified minimum support ξ, an itemset X is a **Frequent Itemset** if $supp(X) \geqq \xi$, where $supp(X)$ is the percentage or absolute number of transactions in D which contain X as a subset.

Problem Definition 2.1 (Frequent Itemset Mining): Let D be a transaction database over a set of distinct items I. Given a user-specified minimum support ξ, the problem of **Frequent Itemset Mining** is to find the complete set of frequent itemsets, i.e. $\{X \mid X \subseteq I$ & $supp(X) \geqq \xi\}$. The complete set of frequent itemsets is denoted as **FI**.

Most of the algorithms for mining frequent itemsets can be described using the subset lattice that was originally introduced by R. Rymon [R92]. This lattice shows how sets of items are completely enumerated in a search space. Assume there is a total lexicographic ordering $\leqq_L$ of all items in the database. This ordering is used to enumerate the item subset lattice (search space). A particular ordering affects the item relationships in the lattice but not its completeness. Figure 2.1 shows a sample of a complete subset lattice for four items. The lattice can be traversed in a breadth-first way, or a depth-first way or some other way according to a heuristic. The problem of finding the frequent itemsets in the database can be viewed as finding a *cut* through this lattice so that all those tree nodes (itemsets) above the *cut* are frequent itemsets, while all those below are infrequent.

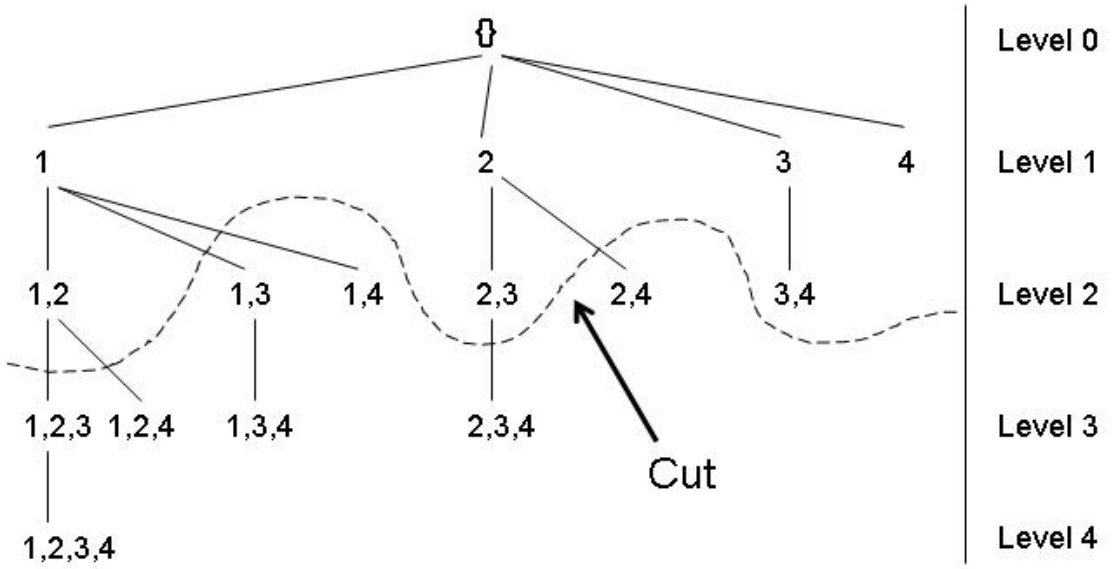


Figure 2.1: Subset Lattice over Four Items for the Given Order of 1, 2, 3, 4

Apriori [AS94] and its variants [BMUT97] [PCY95] [T96] [Z01] employ a bottom-up level-wise search to enumerate every single frequent itemset. These algorithms are based on breadth-first search, i.e. finding all frequent *k*-itemsets before considering (*k*+1)-itemsets. The scalability of such algorithms is greatly compromised by counting all possible 2^k subsets of each frequent *k*-itemset discovered. In 2000, J. Han *et al*. proposed a fundamentally different algorithm, FP-growth [HPY00], which uses the compact frequent pattern tree (FP-tree) structure to record the information of the database and produces frequent itemsets by a pattern fragment growth method to avoid the costly candidate set generation. Mining the FP-tree structure is done recursively by building conditional FP-trees that are of the same order of magnitude in number as the frequent itemsets. However FP-growth requires that the FP-trees fit in the main memory. This makes this algorithm not scalable to sparse and very large databases.

If there exists a frequent *k*-itemset, all its 2^k subsets are frequent. This exponential complexity often makes mining all frequent itemsets impractical when there are very long patterns (30 or 40 or longer) in the data. Algorithms for mining frequent closed itemsets [PHM00] [ZH02] are proposed since they are enough to generate association rules. An itemset is closed if it has no superset with the same support. However, **FCI** could also grow exponentially as **FI**. The problem of frequent closed itemset mining is not the focus of this thesis, but for completeness, we also give the correlative definitions as follows:

Term Definition 2.2 (Frequent Closed Itemset): Let D be a transaction database over a set of distinct items I. Given a user-specified minimum support ξ, an itemset X is a **Frequent Closed Itemset** if $supp(X) \geqq \xi$ and $\forall Y \subseteq I$, if $Y \supset X$ then $supp(Y) < supp(X)$.

Problem Definition 2.2 (Frequent Closed Itemset Mining): Let D be a transaction database over a set of distinct items I. Given a user-specified minimum support ξ, the problem of **Frequent Closed Itemset Mining** is to find the complete set of frequent closed itemsets, i.e. $\{X \mid X \subseteq I \ \& \ supp(X) \geqq \xi \ \& \ \forall Y \subseteq I$, if $Y \supset X$ then $supp(Y) < supp(X)\}$. The complete set of frequent closed itemsets is denoted as **FCI**.

Because of exponential time complexity of **FI** and **FCI** mining, much recent research has turned to mining maximal frequent itemsets. A frequent itemset is called a maximal frequent itemset if it has no superset that is frequent. The set **MFI** is orders of magnitude smaller than the set **FCI**, and in many applications **MFI** is adequate to generate interesting patterns. **MFI** can be used to generate **FI** with a simple generation algorithm. Correlative definitions are given as follows:

Term Definition 2.3 (Maximal Frequent Itemset): Let D be a transaction database over a set of distinct items I. Given a user-specified minimum support ξ, an itemset X is a **Maximal Frequent Itemset** if $supp(X) \geqq \xi$ and $\forall Y \subseteq I$, if $Y \supset X$ then $supp(Y) < \xi$.

Problem Definition 2.3 (Maximal Frequent Itemset Mining): Let D be a transaction database over a set of distinct items I. Given a user-specified minimum support ξ, the problem of **Maximal Frequent Itemset Mining** is to find the complete set of maximal frequent itemsets, i.e. $\{X \mid X \subseteq I \ \& \ supp(X) \geqq \xi \ \& \ \forall Y \subseteq I$, if $Y \supset X$ then $supp(Y) < \xi\}$. The complete set of maximal frequent itemsets is denoted as **MFI**.

Nevertheless, the true focus in this thesis is another much smaller set of itemsets, whose number is usually under several hundred, may even be 1. They are longest frequent itemsets. An itemset is called a longest frequent itemset if it contains the maximum number of items in **FI**. Due to the largely reduced number, mining longest frequent itemsets can be extremely fast. The motivation of finding longest frequent itemsets has been described in Section 1.3. We give the formal definitions as follows:

Term Definition 2.4 (Longest Frequent Itemset): Let D be a transaction database over a set of distinct items I. Given a user-specified minimum support ξ, an itemset X is a **Longest Frequent Itemset** if $supp(X) \geqq \xi$ and $\forall Y \subseteq I$, if $supp(Y) \geqq \xi$ then $|Y| \leqq |X|$, where $|Y|$ and $|X|$ are the number of items contained in Y and X respectively.

Problem Definition 2.4 (Longest Frequent Itemset Mining): Let D be a transaction database over a set of distinct items I. Given a user-specified minimum support ξ, the problem of **Longest Frequent Itemset Mining** is to find the complete set of longest

frequent itemsets, i.e. $\{X \mid X \subseteq I \ \& \ supp(X) \geqq \xi \ \& \ \forall Y \subseteq I$, if $supp(Y) \geqq \xi$ then $|Y| \leqq |X|\}$. The complete set of longest frequent itemsets is denoted as **LFI**.

There may exist more than one longest frequent itemset with the same size. Apparently any longest frequent itemset is a maximal frequent itemset. Thus we have the following relationship: **LFI** $\subseteq$ **MFI** $\subseteq$ **FCI** $\subseteq$ **FI**. Our goal in this thesis is to find **LFI** efficiently.

2.2 Algorithms for Mining Frequent Itemsets

Finding frequent itemsets plays an important role in the data mining field since many data mining problems require this step, such as the discovery of association rules [AS94] [HGN00], data correlations, sequential or multi-dimensional patterns [P01] [PHP01] [SA96] [Z01]. In 1993, Agrawal *et al*. [AIS93] first proposed the problem of finding frequent itemsets in their association rule model and support confidence framework. In this section, we will review some famous algorithms in this domain.

2.2.1 Apriori

As mentioned before, Apriori [AIS93] [AS94] is a classic algorithm for finding frequent itemsets, since most of algorithms are variants of Apriori. It uses frequent itemsets at level k to explore those at level (k+1). In the process of exploring each level, one full scan of the database is required, and a candidate set of frequent itemsets is constructed with the number of occurrences of each candidate itemset being counted. The frequent itemsets are then generated from the candidate itemsets with support no less than the pre-specified minimum support ξ. To improve the efficiency, the Apriori heuristic, i.e.

all nonempty subsets of a frequent itemset must also be frequent, prunes unpromising candidates to narrow down the search space.

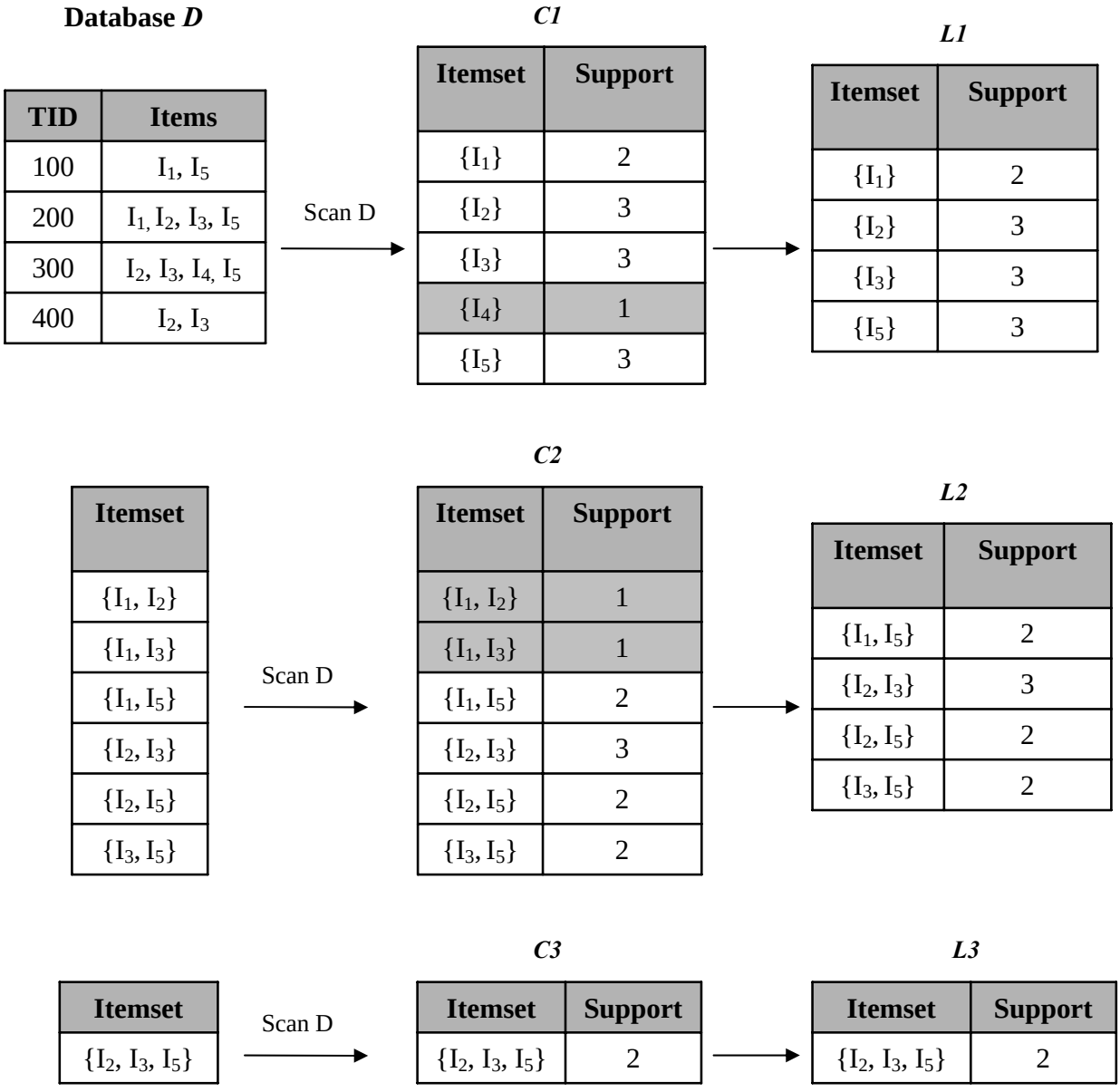


Figure 2.2: An Example of Apriori Algorithm

To better understand how Apriori works, let's consider an example given in Figure

2.2. There are four transactions with five distinct items and the minimum support ξ is supposed to be 2. In the first database scan, Apriori counts the occurrences of each item by scanning all the transactions and the set of candidate frequent *1*-itemsets *C1* is generated. After checking the supports of each candidate, we find the candidate *1*-itemset $\{I_4\}$ is not a frequent itemset because its support is smaller than ξ. The set of frequent *1*-itemsets *L1* is generated by eliminating $\{I_4\}$ from *C1*. Next, Apriori uses *L1* $\rhd\lhd$ *L1* to generate the set of candidate *2*-itemsets *C2*, where $\rhd\lhd$ is a join operation like in relational database. Then the database is scanned the second time and the occurrences of each candidate are counted. The set of frequent *2*-itemsets *L2* is then generated by eliminating infrequent candidates $\{I_1, I_2\}$ and $\{I_1, I_3\}$. *C3* is generated by joining *L2* with itself, i.e. *L2* $\rhd\lhd$ *L2*. Note there is only one candidate itemset in *C3*, some unpromising candidates, such as $\{I_1, I_2, I_5\}$, are pruned by Apriori heuristic because one of its subset $\{I_1, I_2\}$ is not a frequent itemset. After the third scan of the database, *L3* is discovered with one itemset $\{I_2, I_3, I_5\}$. Since there is only one itemset in *L3*, no candidate *4*-itemset can be formed, then the process of frequent itemset mining ends.

2.2.2 FP-growth

FP-growth [HPY00] is a fundamentally different algorithm from the Apriori-like algorithms. The efficiency of Apriori-like algorithms suffers from the exponential enumeration of candidate itemsets and repeated database scans at each level for support check. To diminish these weaknesses, the FP-growth algorithm finds frequent itemsets without candidate set generation and records the database into a compact FP-tree structure to avoid repeated database scanning.

Due to the savings of storing the database in the main memory, the FP-growth algorithm achieves great performance gains against Apriori-like algorithms. However it requires that the FP-trees fit in the main memory. This makes this algorithm not scalable to very large databases.

The FP-tree structure and the FP-growth algorithm are the main bases of our algorithm, and details about them will be presented in Chapter 3.

2.2.3 VIPER

The Apriori and FP-growth algorithms described above are based on the horizontal format of the database representation, in which a transaction is represented as a list of items which occur in this transaction. An alternative way to represent a database is in vertical format, in which each item is associated with a set of transaction identifiers (TIDs) that include the item. Vertical representation has an advantage of performing support counting efficiently.

Database *D*

TID	Items
100	I_1, I_5
200	I_1, I_2, I_3, I_5
300	I_2, I_3, I_4, I_5
400	I_2, I_3

I_1	I_2	I_3	I_4	I_5
100	200	200	300	100
200	300	300		200
	400	400		300

TID of $\{I_1, I_2\}$ = | I_1 | |---| | 100 | | 200 | **AND** | I_2 | |---| | 200 | | 300 | | 400 | → | $\{I_1, I_2\}$ | |---| | 200 |

Figure 2.3: Vertical Database Representation

Let's refer to Figure 2.3. Left is the horizontal representation of a sample database with four transactions, here we transform it to its vertical format on the right. Each item is associated with a unique TID-list, and the support of an item can simply be obtained from calculating the cardinality of its TID-list. For example, there are 2 TIDs in the TID-list of $\{I_1\}$, so that the support of $\{I_1\}$ is 2, suppose the minimum support ξ is equal to 2, we can conclude that $\{I_1\}$ is a frequent itemset, while $\{I_4\}$ is infrequent for the cardinality of its TID-list is only 1. Generating the TID-list of a $(k+1)$-itemset ($k \geqq 1$) can be simply conducted by employing *AND* operation to the TID-lists of its two different k-subsets, which extracts the common entries in the two lists. For example, the TID-list of $\{I_1, I_2\}$ can be obtained by TID-list of $\{I_1\}$ *AND* TID-list of $\{I_2\}$, which returns the common transactions containing both I_1 and I_2, and only transaction 200 is found contained in the TID-list of $\{I_1, I_2\}$.

VIPER [SHS00] is an algorithm based on the vertical format that can sometimes outperform even the optimal method using a horizontal layout. It uses TID-bitvector rather than TID-list due to the large space cost of TID-list with each entry taking up $\log_2 N$ bits, where N is the number of transactions. For those large databases with high-support itemsets, the considerable space cost would be a problem. VIPER solves this problem by compressing TID-list into the TID-bitvector structure which represents each transaction by one bit, which indicates whether the transaction occurs in the TID-list or not. VIPER uses a vertical bitvector with compression to store intermediate data during algorithm execution, while counting is still performed by the *AND* operation like the vertical TID-list approach [BCG01].

2.3 Algorithms for Mining Maximal Frequent Itemsets

A big problem of mining frequent itemsets is that in many databases with long patterns, it would be computationally infeasible to enumerate all possible 2^k subsets of a frequent k-itemset (k can easily be 30 or 40 or longer). Algorithms for mining frequent closed itemsets are proposed since they are enough to generate association rules. However, **FCI** could also grow exponentially as **FI**. The set **MFI** is orders of magnitude smaller than the set **FCI**, and in many applications **MFI** is adequate to generate interesting patterns. In the following, we will introduce some famous algorithms on mining **MFI**. Mining **FCI** is not our focus, for more information about this topic, please refer to [PHM00] [ZH02].

2.3.1 Pincer-Search

The Pincer-Search algorithm [LK98], which is designed for mining maximal frequent itemsets, combines both the bottom-up and top-down searches when traversing the itemset lattice in breadth-first order. While the major search direction is still bottom-up, a restricted search is conducted in the top-down direction for early pruning of candidates that would normally be encountered in the bottom-up search. It maintains a candidate set of maximal patterns to help eliminating the non-maximal itemsets, and consequently the number of database scans is reduced. Because some candidates in the bottom-up search are pruned in advance, a recovery operation is required to restore some wrongly pruned itemsets to the current candidate set in the bottom-up search, which could be a drawback for Pincer-Search. Also, the overhead of maintaining the maximal candidate set can be very high.

2.3.2 Max-Miner

Max-Miner [B98] is another algorithm for mining maximal frequent itemsets. It employs a breadth-first traversal of the search space as well, but also uses lookahead to prune branches from the itemset lattice by quickly identifying long frequent itemsets. To increase the effectiveness of this pruning, Max-Miner orders the items in frequency ascending order to assure the most frequent items to appear in the most candidate groups [B98], since those items are more likely to be part of long frequent itemsets. However, Max-Miner uses a breadth-first approach to limit the number of passes over a database, which compromises the effectiveness of lookahead pruning. In general, lookahead is more suitable in a depth-first approach, since useful longer frequent itemsets can be discovered earlier than some shorter ones.

2.3.3 DepthProject

All of the following algorithms are designed for mining maximal frequent itemsets in a depth-first way. The DepthProject algorithm searches the itemset lattice in a depth-first manner to find maximal frequent itemsets. The lattice is called a lexicographic tree in [AAP00]. It also uses dynamic reordering of children nodes to reduce the size of the search space by trimming infrequent items out of each node's tail. Superset pruning is employed to discover some (k+1)-itemsets before generating all k-itemsets. Also, an improved counting method and a projection mechanism reduce the size of the database. However, DepthProject returns a superset of **MFI** and needs a post-pruning method to remove non-maximal itemsets.

2.3.4 MAFIA

MAFIA uses a vertical format to represent the database, which allows efficient support counting and is said to enhance the effect of lookahead pruning in general [BCG01]. MAFIA compresses and projects the bitmaps to improve performance. In addition, it uses three pruning strategies to remove non-maximal itemsets. The first is lookahead pruning, which was first used in Max-Miner. The second is to check if a candidate set is subsumed by an existing maximal set. If so, it could be eliminated before counting its support. The last technique checks if $t(X) \subseteq t(Y)$, where X and Y are itemsets and $t(X)$ and $t(Y)$ are the set of transactions that contain X and Y respectively. If so, X is considered together with Y for extension. MAFIA mines a superset of **MFI** and requires a post-pruning step to eliminate non-maximal patterns. Moreover, MAFIA assumes that the entire database and all data structures it uses can completely fit in main memory, which limits its application on some huge databases.

2.3.5 GenMax

Unlike DepthProject and MAFIA, GenMax returns the exact **MFI**. It is an algorithm that utilizes a backtracking search for efficiently enumerating all maximal patterns [GZ01]. It uses a novel technique, called progressive focusing, for superset checking. It maintains a local set of relevant maximal itemsets called LMFI. A newly generated candidate is checked in LMFI instead of in the full **MFI** set found so far. This speeds up the process of superset checking. As well, GenMax represents the database in a vertical TID-list format like VIPER [SHS00] and uses diffset [ZG01] propagation to perform fast support counting.

2.3.6 FPMAX

FPMAX [GZ03], which is an extension of the FP-growth algorithm, also finds the exact **MFI**. As with FP-growth, the highly compact FP-tree structure is used to store the information concerning frequent items. By adopting a pattern fragment growth method, it avoids costly candidate generation-and-test. A novel Maximal Frequent Itemset tree (MFI-tree) structure is utilized to keep track of all maximal frequent itemsets. This structure makes FPMAX perform subset checking more efficiently. Experimental results show that FPMAX has comparable performance with MAFIA and GenMax and also it has good scalability.

Chapter 3

Mining LFI with FP-tree

In this chapter, we first introduce the FP-tree structure and the FP-growth algorithm [HPY00], upon which our algorithms are based. Then we describe our variant of the FPMAX algorithm. After discussing the methods to prune the search space, we present the LFIMiner algorithm, which integrates these methods to realize performance gains. The LFIMiner_ALL algorithm is presented at the end of this chapter.

3.1 FP-tree and FP-growth Algorithm

The frequent pattern tree (FP-tree) is a novel compact data structure used by the FP-growth algorithm to store the information about frequent itemsets in a database. The frequent items of each transaction are inserted into the tree in their frequency descending order. Compression is achieved by building the tree in such a way that overlapping transactions are represented by sharing common prefixes of the corresponding branches. A header table is associated with the FP-tree for facilitating tree traversal. Items are sorted in the header table in frequency descending order. Each row in the header table represents a frequent item, containing a *head of node-link* that links all the corresponding nodes in the tree.

TID	Itemset
100	I_3, I_5
200	I_1, I_2, I_3, I_5
300	I_2, I_3, I_4, I_5
400	I_2, I_3
500	I_1, I_2
600	I_2, I_6
700	I_2
800	I_1, I_3, I_5
900	I_1, I_3, I_4

Table 3.1: Example of Transaction Database

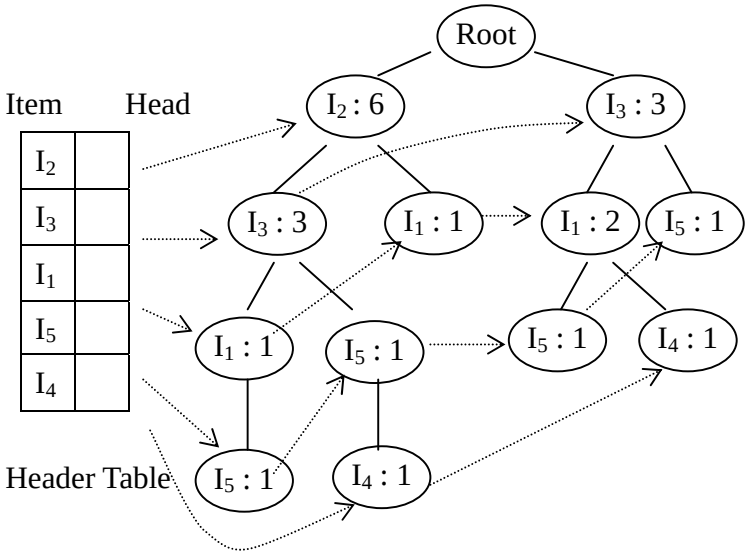


Figure 3.1: FP-tree for the Database in Table 3.1

Unlike Apriori-like algorithms which need several database scans, the FP-growth algorithm needs only two database scans. The first scan collects the set of frequent items. The second scan constructs the initial FP-tree, which records the information of the original database. Table 3.1 is an example database. After the first scan, the set of

frequent items, {(I_2 : 6), (I_3 : 6), (I_1 : 4), (I_5 : 4), (I_4 : 2)} (sorted in frequency descending order, minimum support is 2), is derived. In the second scan, each transaction is inserted into the tree. The scan of the first two transactions extracts their frequent items and constructs the first two branches of the tree: {(I_3 : 1), (I_5 : 1)} and {(I_2 : 1), (I_3 : 1) , (I_1 : 1), (I_5 : 1)}. For the third transaction, since its frequent item list {I_2, I_3, I_5, I_4} shares a common prefix {I_2, I_3} with the existing path {I_2, I_3, I_1, I_5}, the count of each node along the prefix is incremented by 1, and one new node (I_5 : 1) is created and linked as a child of node (I_3 : 2) and another new node (I_4 : 1) is created and linked as a child of node (I_5 : 1). Figure 3.1 shows the initial FP-tree constructed after scanning all the transactions.

The FP-growth algorithm is based on the following principle: Let X and Y be two itemsets in database D, B be the set of transactions in D containing X. Then the support of $X \cup Y$ in D is equivalent to the support of Y in B. B is called the *conditional pattern base* of X. Given an item in the header table, the FP-growth algorithm constructs a new FP-tree according to its conditional pattern base, and mines this FP-tree recursively. Let's examine the mining process based on the FP-tree shown in Figure 3.1. We start from the bottom of the header table. For item I_4, it derives a frequent itemset (I_4 : 2) and two paths in the FP-tree: {(I_2 : 1), (I_3 : 1), (I_5 : 1)} and {(I_3 : 1), (I_1 : 1)}, which constitute I_4's conditional pattern base. An FP-tree constructed from this conditional pattern base, called I_4's *conditional FP-tree*, has only one branch {(I_3 : 2)}. Therefore only one frequent itemset (I_3I_4 : 2) is derived. The exploration for frequent itemsets associated with item I_4 is terminated. Then one can continue to explore item I_5. For more information about the FP-tree and FP-growth algorithm, please refer to [HPY00].

3.2 The FPMAX_LO Algorithm

Based on the FP-growth algorithm, one can find all frequent itemsets. But in order to solve our problem, some modifications are required to guarantee that the frequent itemset generated by our algorithm is the longest frequent itemset. We constructed a simple version by extending the FP-growth algorithm. When we constructed our variant of the FPMAX algorithm, we found that they are completely identical. It is not surprising, because the FPMAX algorithm is also an extension of the FP-growth algorithm. For consistency, we use a uniform name: FPMAX_LO ("Longest Only").

The FPMAX_LO algorithm is shown in Figure 3.2. Like FP-growth, algorithm FPMAX_LO is recursive. The initial FP-tree constructed from the two scans of the database is passed on as the parameter of the first call of the algorithm. An item list *Head*, initialized to be empty, contains the items whose conditional FP-tree will be constructed from its conditional pattern base and will then be mined recursively. Before recursive call to FPMAX_LO, we already know that the combination set of *Head* and the items in the FP-tree is longer than the longest frequent itemset found so far (guaranteed by line (7)). Thus if there is only one single path in the FP-tree, the items in this path, together with *Head*, constitute a longer frequent itemset. If the FP-tree is not a single-path tree, then for each item in the header table, append the item to *Head*, construct the conditional pattern base of the new *Head*, and check in line (7) whether the combination set of *Head* with all frequent items *Tail* in the conditional pattern base is longer than the longest frequent itemset so far. If yes, we construct the conditional FP-tree based on the conditional pattern base and explore this tree recursively.

Input: *T*: an FP-tree
Global:
lfi: the longest frequent itemset found so far
Head: a list of items
Tree: the initial FP-tree
Output: The *lfi* that is a longest frequent itemset
Method: Call **FPMAX_LO** (*Tree*).
Procedure **FPMAX_LO** (*T*) {
(1) IF *T* only contains a single path *P*
(2) THEN update *lfi* with *Head* ∪ *P*;
(3) ELSE FOR EACH item *i* in header table of *T* DO {
(4) Append *i* to *Head*;
(5) Construct *Head*'s conditional pattern base;
(6) *Tail* = {frequent items in *Head*'s conditional pattern base};
(7) IF Length(*Head* ∪ *Tail*) > Length(*lfi*)
(8) THEN {
(9) Construct *Head*'s conditional FP-tree T_{Head};
(10) **FPMAX_LO** (T_{Head}); }
(11) Remove *i* from *Head*. } //end of for each
} // end of procedure

Figure 3.2: The FPMAX_LO Algorithm

3.3 Pruning Away the Search Space

The FPMAX_LO algorithm runs without any pruning. To realize better performance, we added pruning. The FPMAX_LO algorithm has three main operations: construct the conditional pattern base, find all frequent items in the conditional pattern base, and construct the conditional FP-tree. By in-depth study, we found pruning techniques related

to these three operations. For the first operation, conditional transactions that are not long enough cannot be useful for generating a longer frequent itemset and should be removed. For the second operation, conditional transactions which don't have enough frequent items cannot contribute to forming a longer frequent itemset and should be trimmed. For the last one, we can reorder items by descending order of frequency in each FP-tree, which often makes the FP-trees more compact and thus prunes the search space. We will elaborate on each of these pruning strategies in the following paragraphs.

3.3.1 Conditional Pattern Base Pruning (CPP)

The Conditional Pattern Base Pruning (CPP), as presented in Figure 3.4, is applied during construction of the conditional pattern base. It prunes conditional transactions, and at the same time tries to find a frequent itemset longer than the longest frequent itemset found so far (referred as *lfi* below). Straightforwardly, any conditional transaction t belonging to the conditional pattern base should satisfy $Head \cup t$ *is longer than lfi*; otherwise it should be pruned because it cannot contribute to forming a longer itemset. This strategy has been discussed in [WZ02]. For instance, in Figure 3.3, suppose that the length of *lfi* is 3 and *Head* is $\{I_5, I_4\}$. We are currently looking for a frequent itemset longer than 3. With the FPMAX_LO algorithm, two conditional transactions $\{(I_1 : 2), (I_2 : 2)\}$ and $\{(I_1 : 1)\}$ constitute the conditional pattern base of *Head*. However, the conditional transaction $\{(I_1 : 1)\}$ cannot contribute to forming a frequent itemset longer than 3 because the length of $\{I_5, I_4\} \cup \{I_1\}$ is only 3. CPP cuts such not-long-enough conditional transactions as $\{(I_1 : 1)\}$. If a conditional transaction t has a sufficient length and, moreover, t is frequent, a longer frequent itemset $Head \cup t$ is discovered. Let's

examine Figure 3.3 again, and suppose that the minimum support is 2. The conditional transaction {(I_1 : 2), (I_2 : 2)} is frequent, so we obtain a longer frequent itemset {I_5, I_4, I_1, I_2} immediately. Due to its larger length than previous *lfi*, {I_5, I_4, I_1, I_2} is expected to be able to prune more not-long-enough conditional transactions.

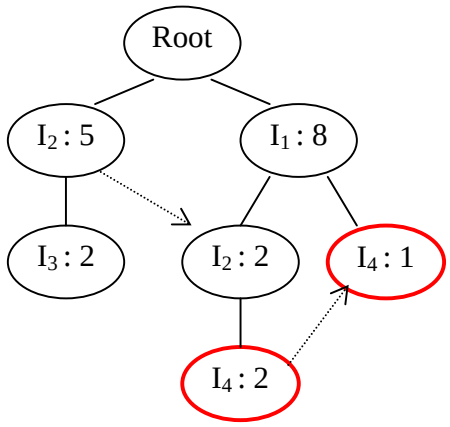


Figure 3.3: An Example of the Conditional Pattern Base Pruning

Input: *HT*: a header table
Global:
lfi: the longest frequent itemset found so far
Head: a list of items
Output: *CPB*: the conditional pattern base of *Head*
Method: Call **ConstructCondPatternBase** (*HT*).
Procedure **ConstructCondPatternBase** (*HT*) {
(1) FOR EACH conditional transaction *t* in *HT* (visiting via node links){
(2) IF Length(*Head* ∪ *t*) > Length(*lfi*)
(3) THEN IF *t* is frequent
(4) THEN update *lfi* with *Head* ∪ *t*;
(5) ELSE insert *t* into *CPB*. } // end of for each
} // end of procedure

Figure 3.4: Construct Conditional Pattern Base

3.3.2 Frequent Item Pruning (FIP)

FIP pruning, as described in Figure 3.6, is applied during the "finding all frequent items in the conditional pattern base" phase. Its pruning principle is that any conditional transaction that contains insufficient frequent items cannot contribute to generating a longer frequent itemset and should be trimmed. It imposes a stricter condition on the screening of conditional transactions than CPP. Let's study an example. In Figure 3.5 (1), there are three conditional transactions in the conditional pattern base. Suppose that *Head* is $\{I_7, I_6\}$, the minimum support is 2, and the length of the longest frequent itemset found so far is 4. In the conditional pattern base, I_5 is not a frequent item, so conditional transaction 300 could be considered as $\{(I_2 : 1), (I_4 : 1)\}$, which is noncontributing because $\{I_7, I_6\} \cup \{I_2, I_4\}$ is no longer than 4. Thus, transaction 300 is eliminated from the conditional pattern base (Figure 3.5 (2)). Since some transactions have been removed from the conditional pattern base, some previously frequent items may become infrequent, and thus some other transactions may become noncontributing. We recursively call the procedure to trim more transactions. Let's continue the example. In Figure 3.5 (2), I_2 and I_4 become infrequent this time. As above, transaction 100 and 200 are removed. Then the conditional pattern base is empty, i.e. we can stop exploring *Head* $\{I_7, I_6\}$.

TID	Conditional Transaction in CPB
100	$\{I_1, I_2, I_3\} : 1$
200	$\{I_1, I_3, I_4\} : 1$
300	$\{I_2, I_4, I_5\} : 1$

(1)

1st call →

TID	Conditional Transaction in CPB
100	$\{I_1, I_2, I_3\} : 1$
200	$\{I_1, I_3, I_4\} : 1$

(2)

2nd call → Ø

Figure 3.5: An Example of Frequent Item Pruning

Input: *CPB*: the conditional pattern base of *Head*
Global:
lfi: the longest frequent itemset found so far
Head: a list of items
Local:
Tail: the set of all frequent items in *CPB*
Deleted: the number of deleted transactions
Output: modified *CPB* and *Tail*
Method: Call **GetFrequentItemsinCPB** (*CPB*).

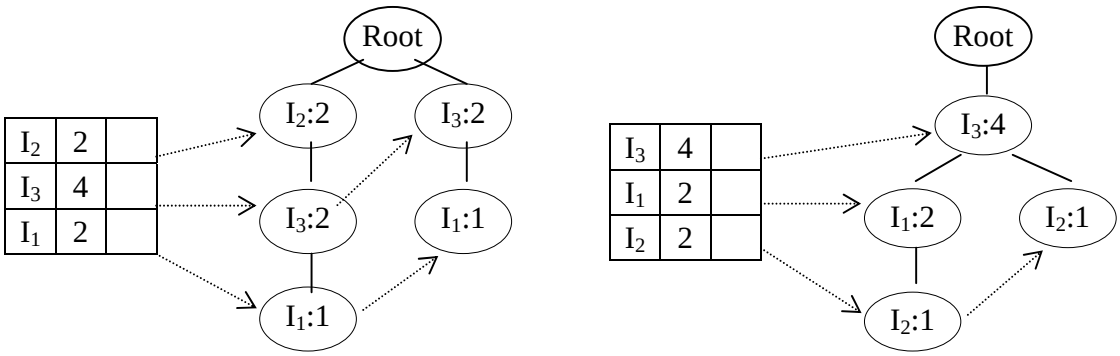
```
Procedure GetFrequentItemsinCPB (CPB) {
(1)  Find all frequent items (Tail) in CPB;
(2)  FOR EACH conditional transaction t in CPB {
(3)        IF Length(Head ∪ {frequent items in t}) ≦ Length(lfi)
(4)        THEN {
(5)              Remove t from CPB;
(6)              Deleted = Deleted + 1; } }      // end of for each
(7)  IF Deleted > 0
(8)  THEN GetFrequentItemsinCPB (CPB);
(9)  ELSE return CPB and Tail.
}    // end of procedure
```

Figure 3.6: Get Frequent Items in Conditional Pattern Base

3.3.3 Dynamic Reordering (DR)

As stated in [HPY00], sorting the items in the header table by descending order of frequency will often increase the compression rate for an FP-tree compared with its corresponding database. The items in transactions will be inserted into FP-tree under their order in the header table, and a sorted header table will keep the nodes of more frequent

items closer to the root, which usually enables more sharing of paths and produces high compression. However, the FP-growth and the FPMAX algorithms only reorder the items in the header table of the initial FP-tree, and follow this order to construct header tables of conditional FP-trees.



(a) Without Dynamic Reordering (b) With Dynamic Reordering

Figure 3.7: Header Table and Conditional FP-tree

In our algorithm, we apply the reordering process to the header tables of all FP-trees, which we expect to make the FP-trees more compact. We dynamically sort items in the header table in descending order of frequency before generating each FP-tree. This "Dynamic Reordering" is also addressed in [CZ03]. Generally, it will improve the performance in both the space (less memory with smaller FP-trees) and time (fewer recursions). Let's study the following example. We refer to the database in Table 3.1 and its corresponding initial FP-tree in Figure 3.1. For item I_5, there are four conditional transactions, {(I_1 : 1), (I_3 : 1), (I_2 : 1)}, {(I_3 : 1), (I_2 : 1)}, {(I_1 : 1), (I_3 : 1)} and {(I_3 : 1}}. The FP-growth algorithm generates the header table as {I_2 : 2, I_3 : 4, I_1 : 2} from top to bottom following the order of items in the header table of the initial FP-tree, and the corresponding conditional FP-tree is shown in Figure 3.7 (1). In contrast, our algorithm

organizes the header table as {I_3 : 4, I_1 : 2, I_2 : 2} from top to bottom after dynamic reordering and constructs the conditional FP-tree as shown in Figure 3.7 (2). There are five nodes in the conditional FP-tree in Figure 3.7 (1) but only four nodes in Figure 3.7 (2), which shows the contribution of Dynamic Reordering in this case.

3.4 The LFIMiner Algorithm

Input: *T*: an FP-tree
Global:
lfi: the longest frequent itemset found so far
Head: a list of items
Tree: the initial FP-tree
Output: The *lfi* that is the longest frequent itemset
Method: Call **LFIMiner** (*Tree*).
Procedure **LFIMiner** (*T*) {
(1) IF *T* only contains a single path *P*
(2) THEN update *lfi* with *Head* ∪ *P*;
(3) ELSE FOR EACH item *i* in header table of *T* DO {
(4) Append *i* to *Head*;
(5) Construct *Head*'s conditional pattern base using CPP;
(6) *Tail* = {frequent items in *Head*'s conditional pattern base} using FIP;
(7) IF Length(*Head* ∪ *Tail*) > Length(*lfi*)
(8) THEN {
(9) Construct *Head*'s conditional FP-tree T_{Head} using DR;
(10) **LFIMiner** (T_{Head}); }
(11) Remove *i* from *Head*. } // end of for each
} // end of procedure

Figure 3.8: The LFIMiner Algorithm

The LFIMiner algorithm, which contains the CPP and FIP pruning methods and Dynamic Reordering, is shown in Figure 3.8. The differences from the FPMAX_LO algorithm are highlighted by underlining.

3.5 The LFIMiner_ALL Algorithm

To find all the longest frequent itemsets in a database, we modified the LFIMiner algorithm a little. Usually, the **LFI** set is orders of magnitude smaller than the **MFI** set. Thus the **LFI** mining process should be quick. The LFIMiner_ALL algorithm is presented in Figure 3.9. The differences from the LFIMiner algorithm are highlighted by underlining. Lines (2)-(7) insert a newly found longest frequent itemset into the **LFI** set. Note the difference between line (12) in Figure 3.9 and line (7) in Figure 3.8. In the LFIMiner_ALL algorithm, the cases where the combination set of *Head* with all frequent items *Tail* in *Head*'s conditional pattern base has equal length with the longest frequent itemsets found so far cannot be neglected as in the LFIMiner algorithm. The CPP and FIP pruning methods were modified accordingly, which are shown in Figure 3.10 and Figure 3.11. We also constructed modified versions of MAFIA_LO and FPMAX_LO that find all the longest frequent itemsets, called MAFIA_LO_ALL and FPMAX_LO_ALL, for performance comparison. These algorithms are presented in Appendix A.

Input: *T*: an FP-tree

Global: *Head*: a list of items; *Tree*: the initial FP-tree

LFIList: the set of longest frequent itemsets found so far

LFILen: the length of longest frequent itemsets found so far

Output: The *LFIList* that is set of all the longest frequent itemsets

Method: Call **LFIMiner_ALL** (*Tree*).

Procedure **LFIMiner_ALL** (*T*) {

(1) IF *T* only contains a single path *P*

(2) THEN IF Length(*Head* ∪ *P*) > *LFILen*

(3) THEN {

(4) Empty *LFIList*;

(5) Insert *Head* ∪ *P* into *LFIList*;

(6) Update *LFILen* with Length(*Head* ∪ *P*); }

(7) ELSE insert *Head* ∪ *P* into *LFIList*;

(8) ELSE FOR EACH item *i* in header table of *T* DO {

(9) Append *i* to *Head*;

(10) Construct *Head*'s conditional pattern base using CPP;

(11) *Tail* = {frequent items in base} using FIP;

(12) IF Length(*Head* ∪ *Tail*) ≧ *LFILen*

(13) THEN {

(14) Construct *Head*'s conditional FP-tree T_{Head} using DR;

(15) **LFIMiner_ALL** (T_{Head}); }

(16) Remove *i* from *Head*. } // end of for each

} // end of procedure

Figure 3.9: The LFIMiner Algorithm

CPB: conditional pattern base; *Head*: a list of items;

Tail: the set of frequent items in *CPB*

LFIList: the set of longest frequent itemsets;

LFILen: the length of longest frequent itemsets

```
Procedure ConstructCPB_ALL (HeaderTable HT) {
(1)   FOR EACH conditional transaction t in HT {
(2)   IF Length(Head ∪ t) ≧ LFILen
(3)   THEN IF t is frequent
(4)         THEN IF Length(Head ∪ t) > LFILen
(5)               THEN {
(6)                     Empty LFIList;
(7)                     Insert Head ∪ t into LFIList;
(8)                     Update LFILen; }
(9)               ELSE Insert Head ∪ t into LFIList;
(10)        ELSE insert t into CPB. } }
```

Figure 3.10: Changed CPP Pruning

```
Deleted: the number of deleted transactions
Procedure GetFrequentItemsinCPB_ALL (CPB) {
(1) Tail = all frequent items in CPB;
(2) FOR EACH conditional transaction t in CPB {
(3)  IF Length(Head ∪ {frequent items in t}) < LFIlen
(4)  THEN {
(5)       Remove t from CPB;
(6)       Deleted = Deleted + 1; } }
(7) IF Deleted > 0
(8) THEN GetFrequentItemsinCPB_ALL (CPB);
(9) ELSE return CPB and Tail. }
```

Figure 3.11: Changed FIP Pruning

Chapter 4

Experimental Results

In this chapter, we present the experimental results for our algorithms. First, we describe an in-depth study on the performance effect of each optimization component. Then we compare the LFIMiner algorithm with the MAFIA_LO and FPMAX_LO algorithms on some real datasets. Finally we present the results concerning the algorithms which find all the longest frequent itemsets.

4.1 Experimental Configuration

All experiments were conducted on a PC with a 2.4 GHz Intel Pentium 4 processor and 512 MB main memory, running Microsoft Windows XP Professional. All codes were compiled using Microsoft Visual C++ 6.0. The MAFIA_LO and MAFIA_LO_ALL algorithms were created by modifying the original source file of MAFIA provided by its authors, and the FPMAX_LO and FPMAX_LO_ALL algorithms were implemented by ourselves. All timing results in the figures are averages of five runs.

We tested the algorithms on the *Mushroom*, *Chess*, *Connect4* and *Pumsb** datasets. *Mushroom* is a benchmark dataset widely used in transaction clustering [GRS99] [WXL99] [XD01]. As described in Chapter 1, **LFI** can be used for transaction clustering, so we chose *Mushroom* into the test. We also used *Chess*, *Connect4* and *Pumsb**, which

contain longer patterns than *Mushroom* to test the efficiency of the algorithms. These datasets have been widely used in frequent itemset mining [AAP00] [B98] [BCG01] [GZ01] [GZ03]. *Mushroom*, *Chess* and *Connect4* are available from UCI Machine Learning Repository [UCIMLR]. *Pumsb** is census data from PUMS (Public Use Microdata Sample). In general, at the higher levels of minimum support, the longest pattern length in these datasets varies between 5-14 items, while for some lower levels of minimum support, the longest patterns have over 20, even 30-40, items. The characteristics of these datasets are shown in Table 4.1.

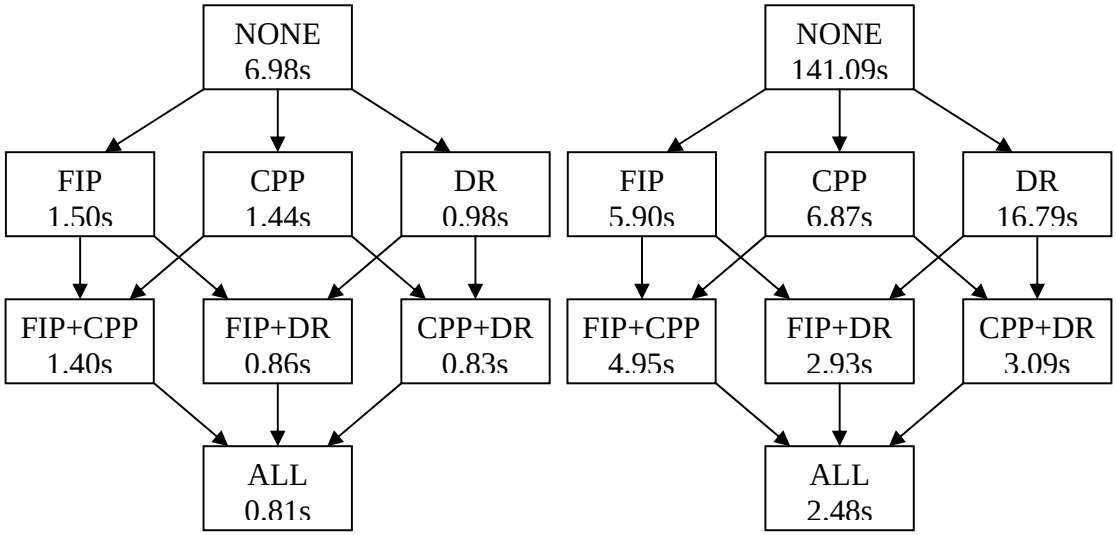
Dataset	File Size (KB)	Num. Trans	Num. Items	ATL*
Mushroom	558	8,124	119	23
Chess	335	3,196	76	37
Connect4	9,039	67,557	129	43
*Pumsb**	11,028	49,046	7,117	50

**ATL*: Average Transaction Length

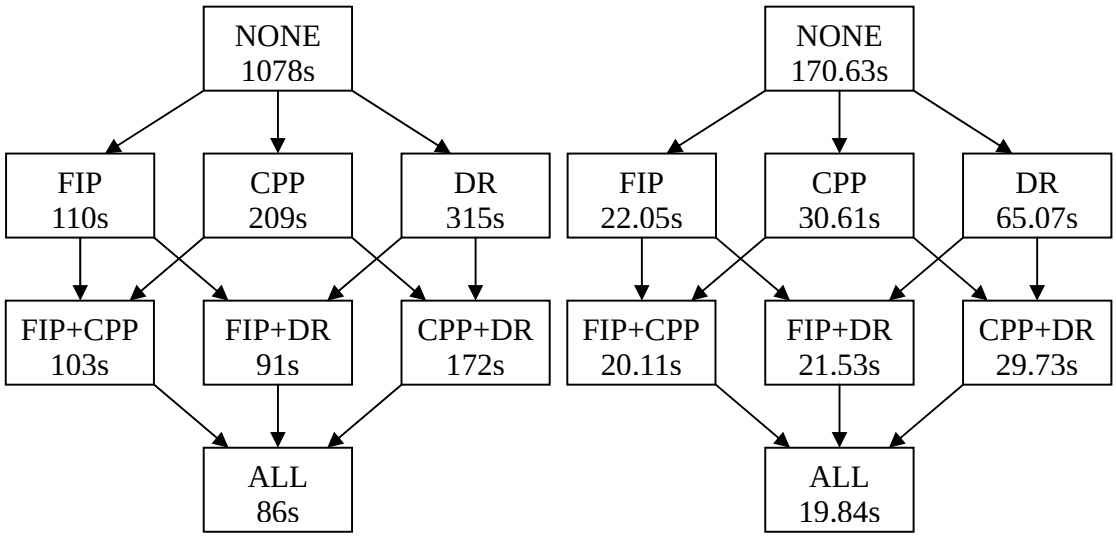
Table 4.1: Dataset Characteristics

4.2 Component Analysis

First, we present a full analysis of component effects on the LFIMiner algorithm. The three main components in our algorithm are: a) CPP pruning, b) FIP pruning and c) Dynamic Reordering (DR). CPP and FIP pruning methods reduce the size of the FP-tree by pruning some noncontributing conditional transactions. Dynamic Reordering reduces the size of the FP-tree by keeping more frequent items closer to the root to enable more sharing of paths.



(1) *Mushroom* at 0.1% minimum support (2) *Chess* at 1% minimum support



(3) *Connect4* at 1% minimum support (4) *Pumsb** at 0.1% minimum support

Figure 4.1: Components' Effects Comparison

The results with different components combination on different datasets are presented in Figure 4.1. The components of the algorithm are represented in a lattice format, in which the running time is shown. We denote the FPMAX_LO algorithm by "NONE", and the FPMAX_LO with each separate component by "FIP", "CPP" and

"DR", respectively. "FIP+CPP" denotes the use of both the FIP and CPP pruning schemes. Finally, the LFIMiner algorithm is denoted by "ALL".

The results consistently show that each component improves performance, and the best results are achieved by combining them together. FIP has the biggest effect among the three components, since it is most likely to trim a large number of candidate transactions by its recursive pruning process. Adding CPP when FIP has already been performed yields little savings, either from FIP to FIP+CPP or from FIP+DR to ALL. Since FIP and CPP both trim conditional transactions, it is not surprising that their efficacy overlaps to some extent. Dynamic Reordering also achieves significant savings.

4.3 Comparison with MAFIA_LO and FPMAX_LO

R. Bayardo at one time implemented a version of Max-Miner that finds the longest frequent itemsets called Max-Miner-LO; however, he didn't describe the algorithmic details in his paper [B98]. And as we know, there are no other existing algorithms to mine the longest frequent itemset. Many algorithms exist for mining **MFI**. [AAP00] shows that DepthProject runs more than an order of magnitude faster than Max-Miner. [BCG01] shows that MAFIA outperforms DepthProject by a factor of three to five. [GZ03] shows that FPMAX achieves comparable performance with MAFIA and GenMax. For performance comparison, we modified the efficient MAFIA and FPMAX algorithms a little to mine the longest frequent itemset. Why did we select these two algorithms? In fact, DepthProject, MAFIA and GenMax share a lot in common: Search the item subset lattice (or lexicographic tree) in a depth-first way, apply lookahead pruning and dynamic reordering to reduce the search space, use a compression technique

for fast support counting. Thus we picked out MAFIA as the representative of the three algorithms. FPMAX is fundamentally different from the above three algorithms and resembles our algorithm because it both employs the same FP-tree structure and extends the same FP-growth algorithm. Thus we also chose FPMAX as a competitor. The FPMAX_LO algorithm has been described in Chapter 3. The MAFIA_LO algorithm will be presented in Appendix A.

Figures 4.2-4.5 illustrate the results of comparing these three algorithms for *Mushroom*, *Chess*, *Connect4* and *Pumsb*,* respectively. In each figure, part (a) shows the running time of the three algorithms. The x-axis is the user-specified minimum support, expressed as a percentage, while the y-axis shows the running time in seconds. Part (b) compares the number of itemsets processed by the FPMAX_LO algorithm and the LFIMiner algorithm. The x-axis is the minimum support, and the y-axis shows the number. Part (c) compares the number of FP-tree nodes created by the FPMAX_LO and LFIMiner algorithms. The x-axis is the minimum support, and the y-axis shows the number.

In general, LFIMiner runs consistently faster than MAFIA_LO and FPMAX_LO, especially when the database is large and the transactions in the database are large (*Connect4* and *Pumsb**). For high levels of support, FPMAX_LO works better than MAFIA_LO, while for low levels of support, MAFIA_LO is more efficient than FPMAX_LO. This can be explained as follows: MAFIA_LO needs a fixed time to convert the database into its vertical format, no matter what the support is. When the support is high, for FPMAX_LO, it will result in a small FP-tree, and thus mining is fast.

MAFIA_LO also mines fast, but the time for database conversion cannot be overlooked. This is reflected in the figures that the time taken by MAFIA_LO for high levels of support changes slightly. A majority of the time is used for database conversion. In contrast, when the support is low, without effective pruning, FPMAX_LO spends considerable time to construct bushy FP-trees. This largely slows down the processing. MAFIA_LO, on the other hand, benefiting from its effective pruning and fast support counting with projected bitmap representation of the database, is not influenced so much. Because the results on all the datasets are similar, we only explain Figure 4.2.

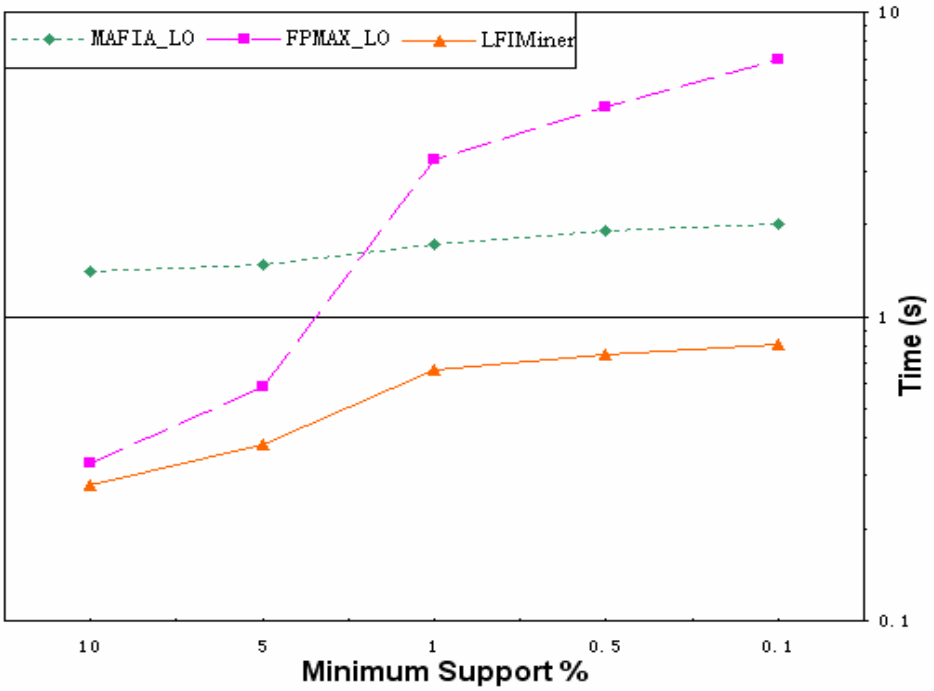


Figure 4.2 (a): Time Comparison on *Mushroom*

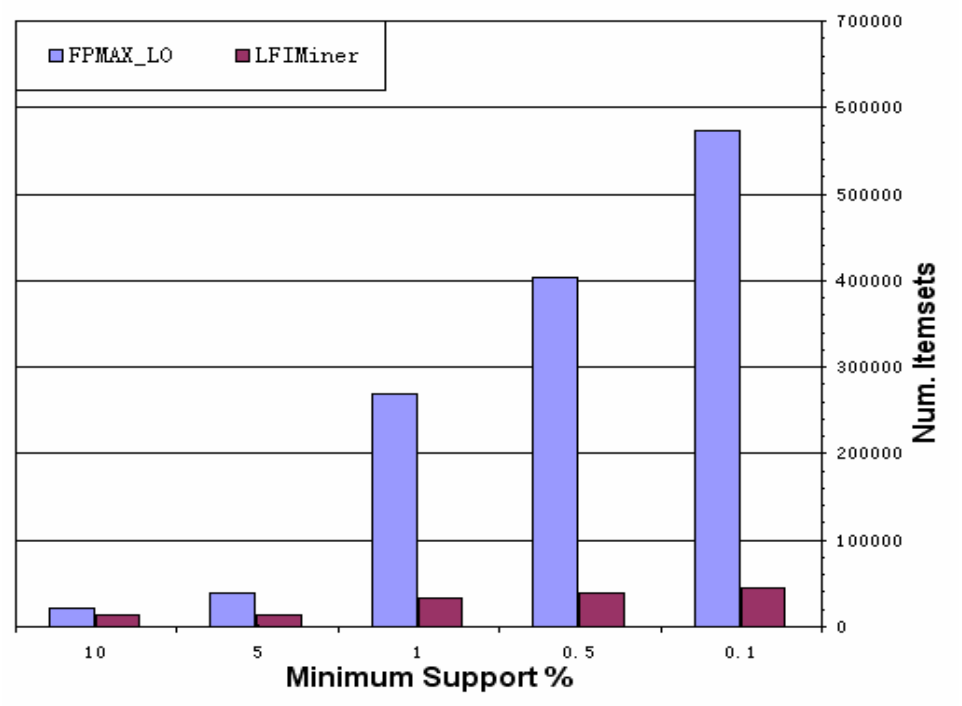


Figure 4.2 (b): Number of Itemsets on *Mushroom*

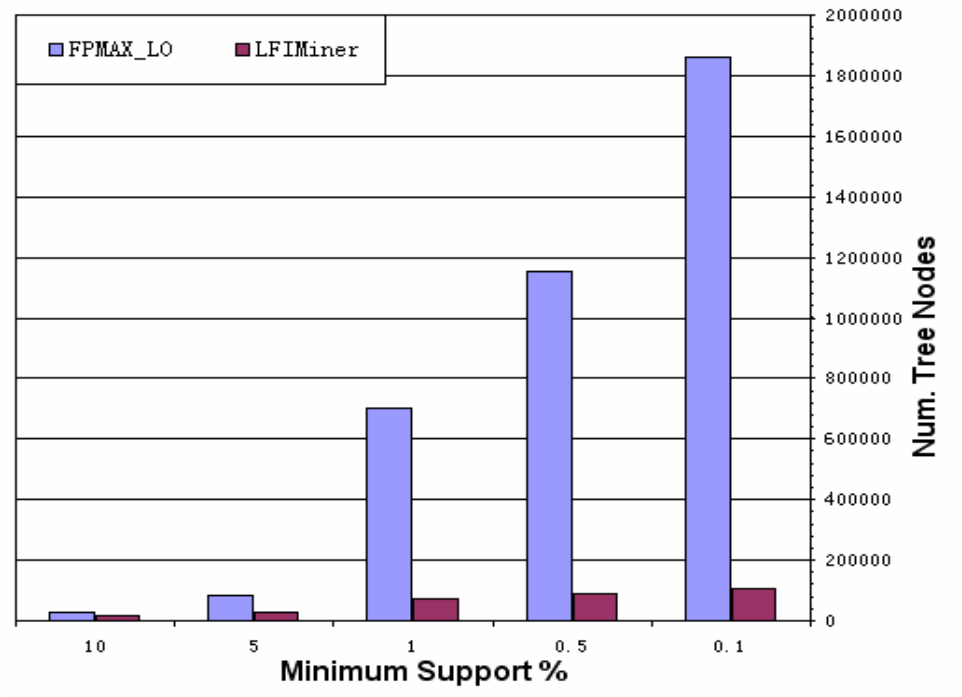


Figure 4.2 (c): Number of Tree Nodes on *Mushroom*

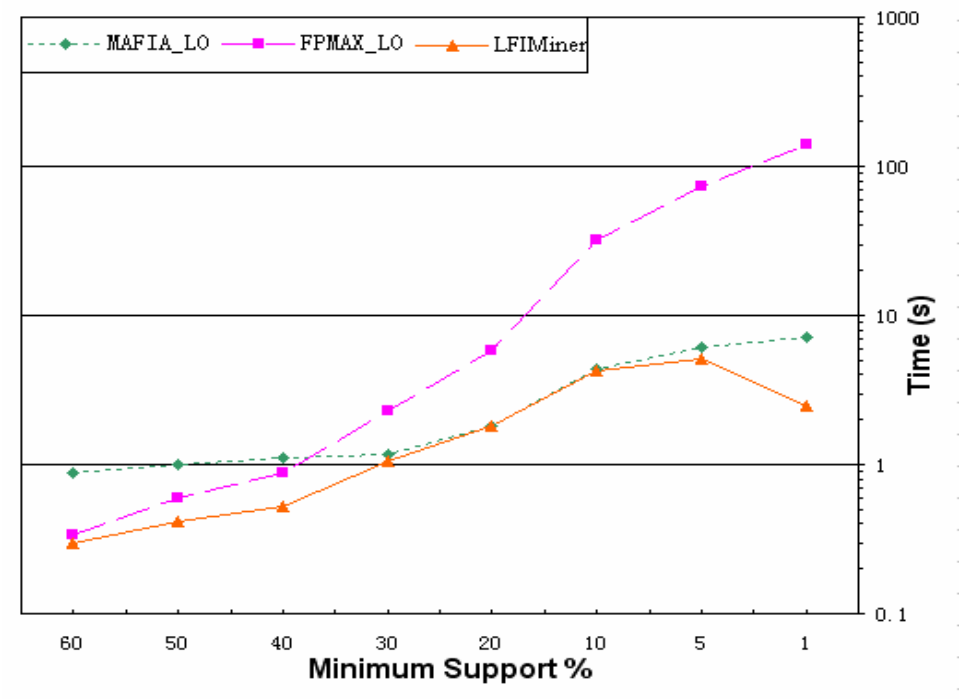


Figure 4.3 (a): Time Comparison on *Chess*

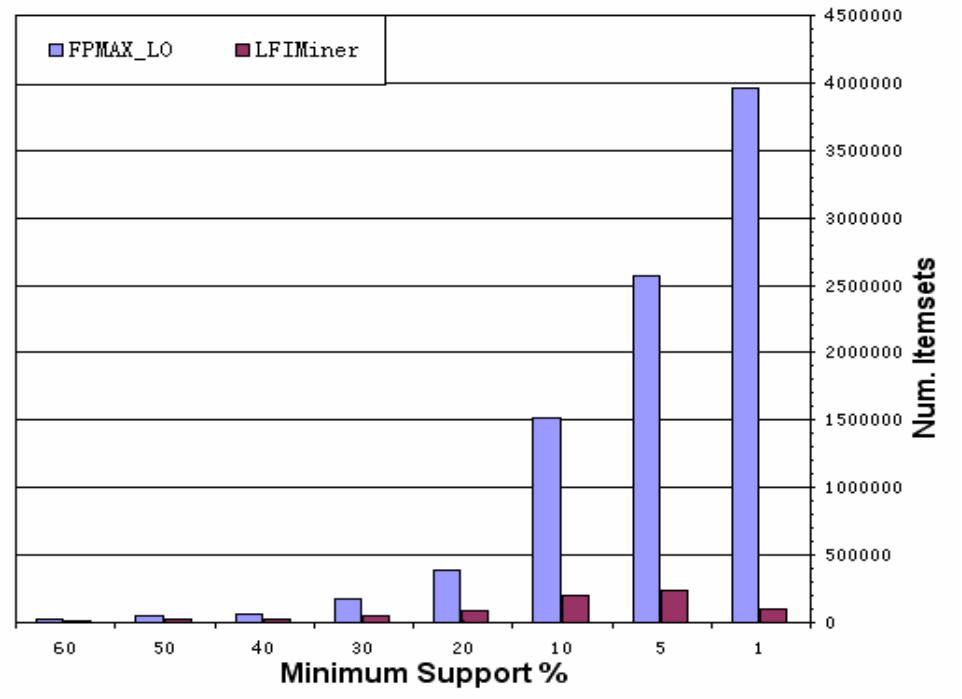


Figure 4.3 (b): Number of Itemsets on *Chess*

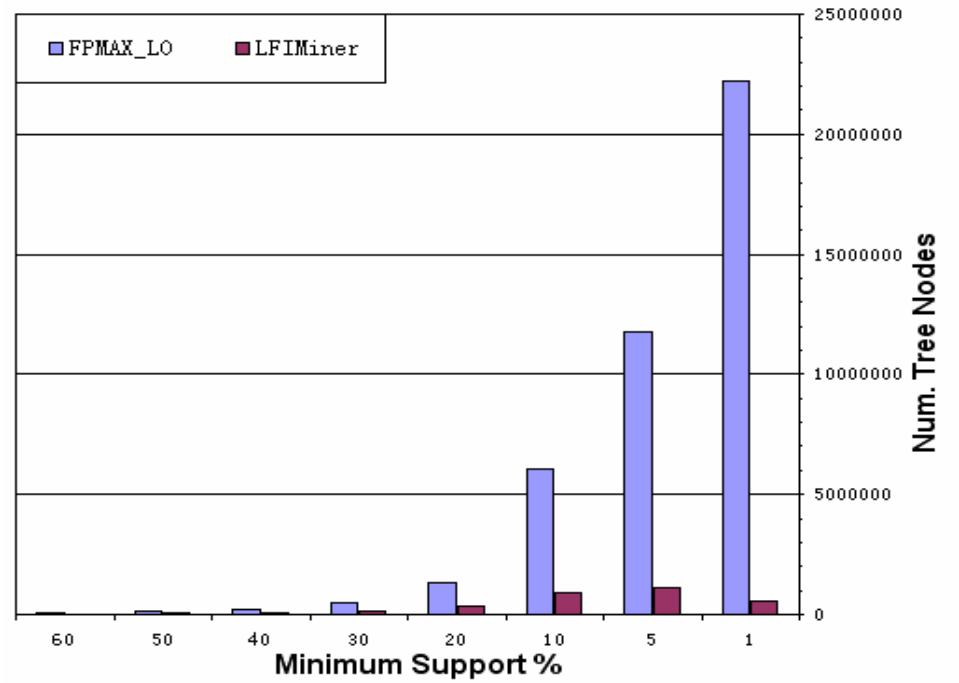


Figure 4.3 (c): Number of Tree Nodes on *Chess*

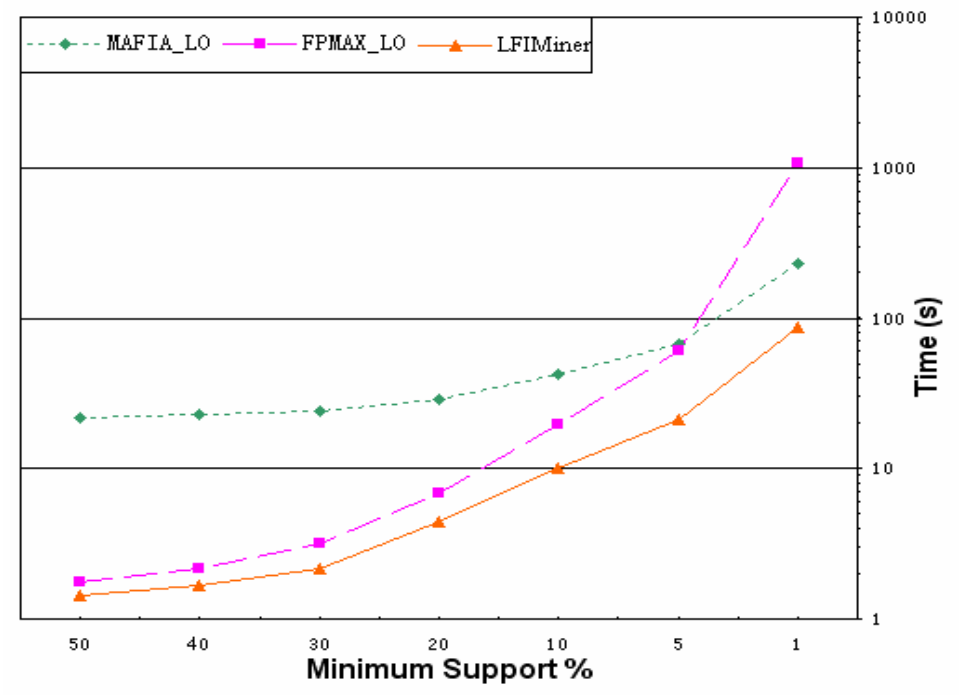


Figure 4.4 (a): Time Comparison on *Connect4*

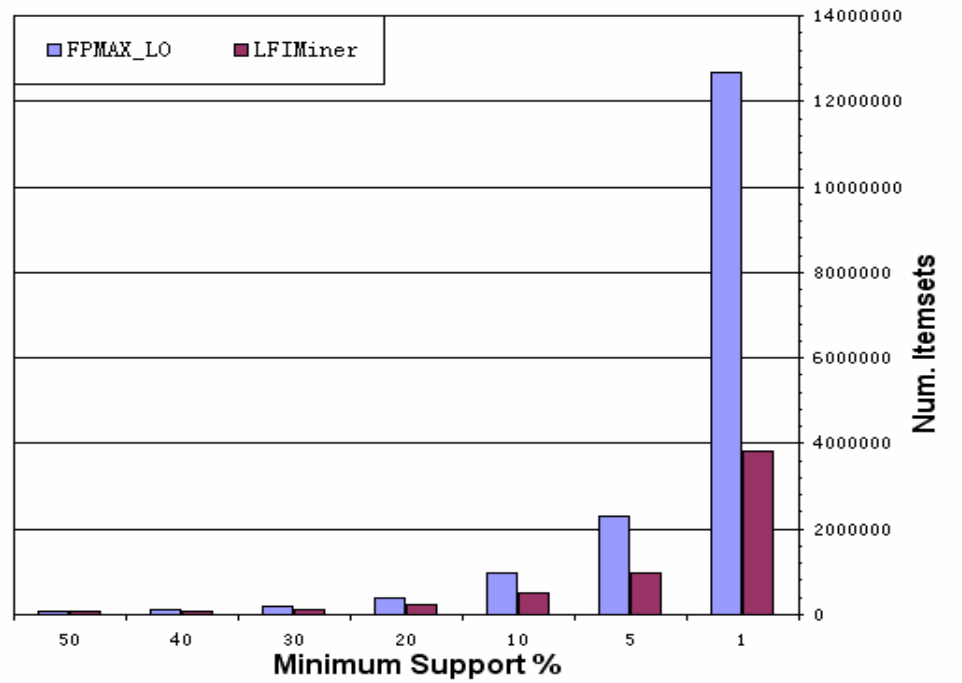


Figure 4.4 (b): Number of Itemsets on *Connect4*

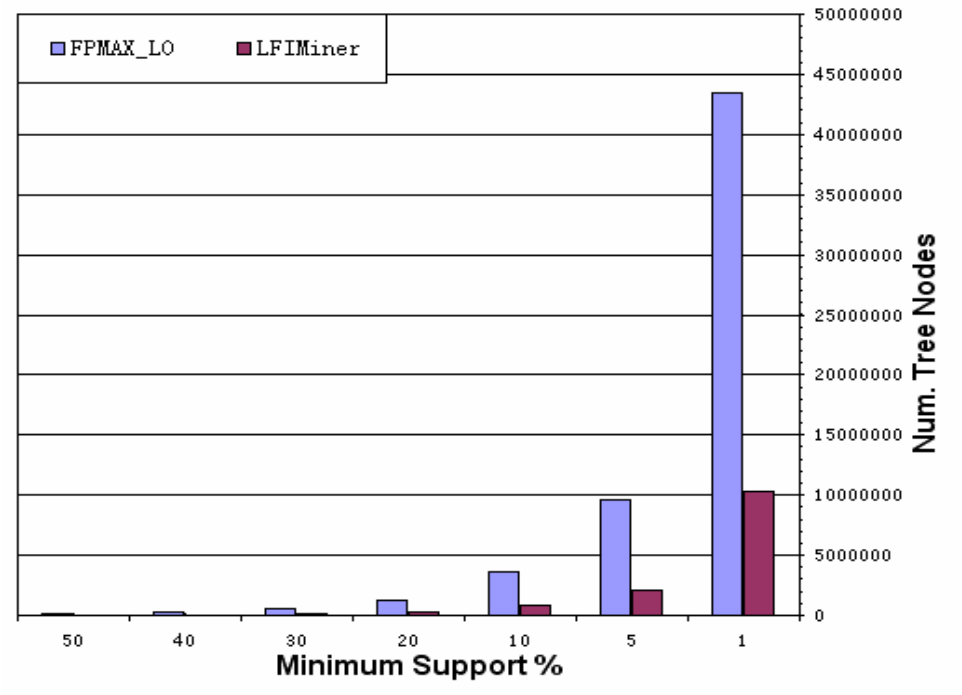


Figure 4.4 (c): Number of Tree Nodes on *Connect4*

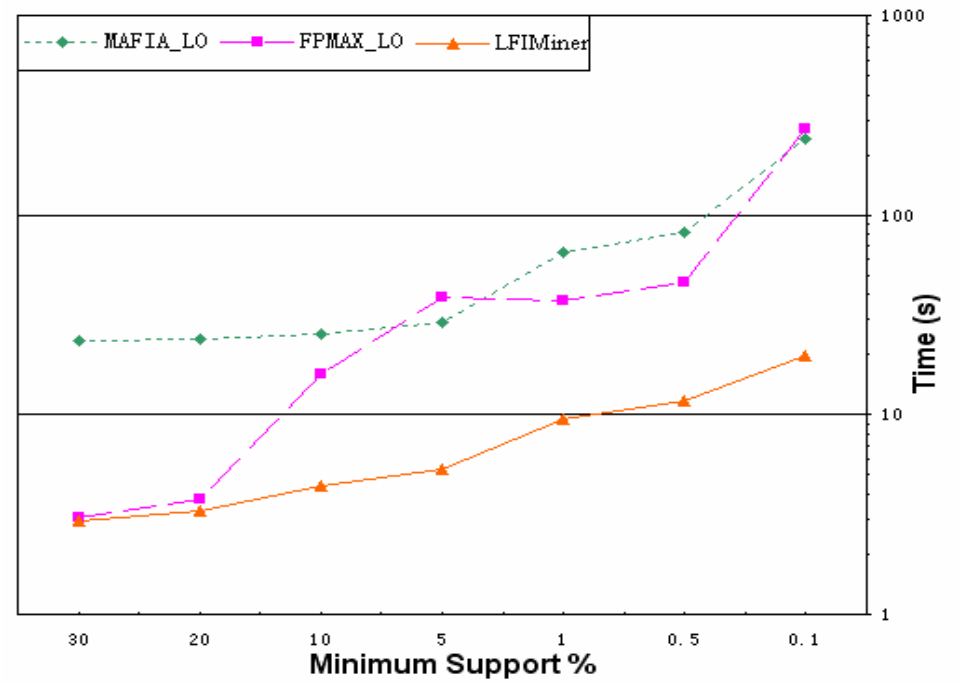


Figure 4.5 (a): Time Comparison on *Pumsb**

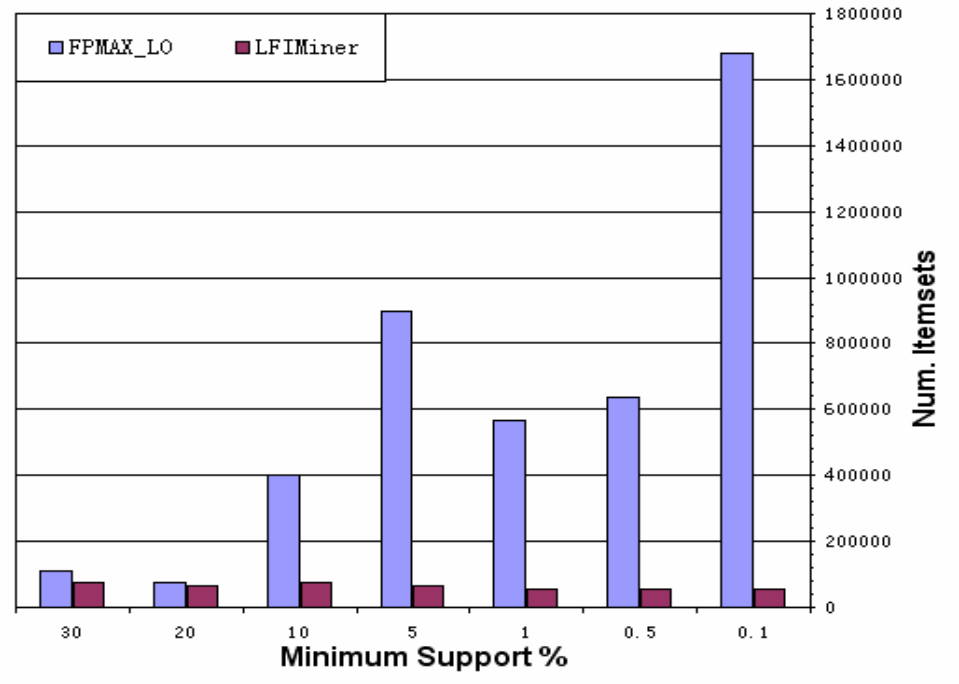


Figure 4.5 (b): Number of Itemsets on *Pumsb**

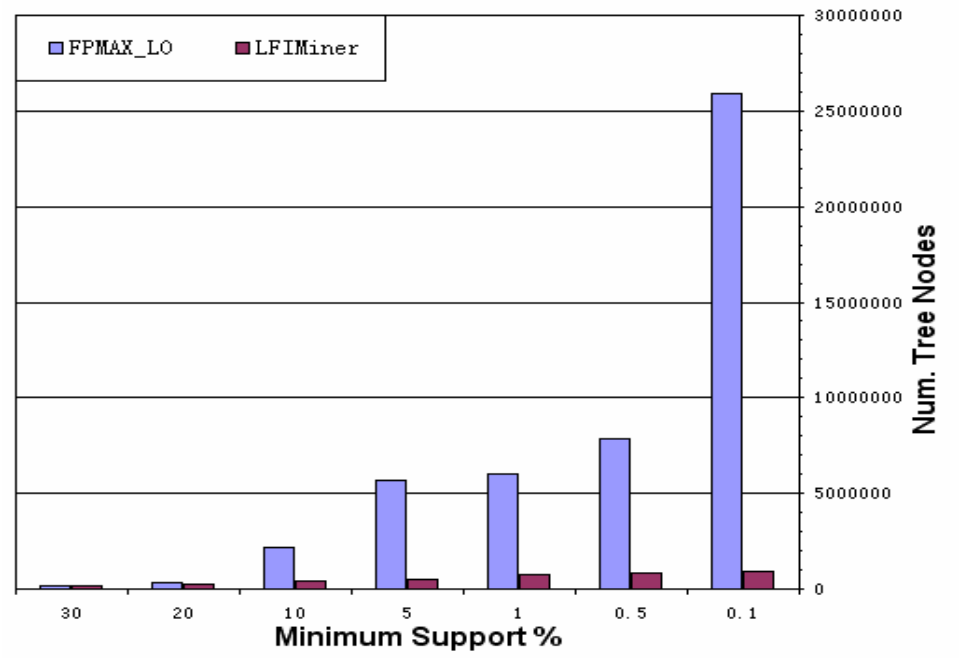


Figure 4.5 (c): Number of Tree Nodes on *Pumsb**

Figure 4.2 shows the results on *Mushroom*. LFIMiner runs faster than MAFIA_LO and FPMAX_LO. MAFIA_LO performs better than FPMAX_LO, as the support is from 1% downwards. As shown in part (b) and part (c), when the support decreases, the number of itemsets processed by FPMAX_LO increases dramatically; this consequently leads to the great increase of tree nodes created while, for LFIMiner, due to effective pruning, the numbers increase slowly. For example, at support 0.1%, the itemsets processed by LFIMiner are only 8% of those processed by FPMAX_LO.

To test the scalability, we ran the three programs on the *Connect4* dataset, which was vertically enlarged by adding transactions into the original dataset. We created new transactions by modeling the distribution of the values of each categorical attribute in the original dataset. In contrast to the *vertical scaling* used in [BCG01], which scaled the

dataset by duplicating the transactions, we created "similar" but not "duplicated" transactions. This is a more realistic way of enlarging the dataset than simply duplicating the dataset.

The support is fixed at 30%. The results are shown in Figure 4.6. All algorithms scale almost linearly with database size; however MAFIA_LO shows a steeper increase than LFIMiner and FPMAX_LO. This is not accidental. As the number of transactions increases, we can expect more similar transactions. For LFIMiner and FPMAX_LO, adding similar transactions to the existing ones will not increase the size of the FP-tree much, while for MAFIA_LO, it increases the cost for support counting more significantly because the bitmaps become long. In addition, because of effective pruning, LFIMiner increases much more slowly than FPMAX_LO. In conclusion, we can say that LFIMiner scales well with database size.

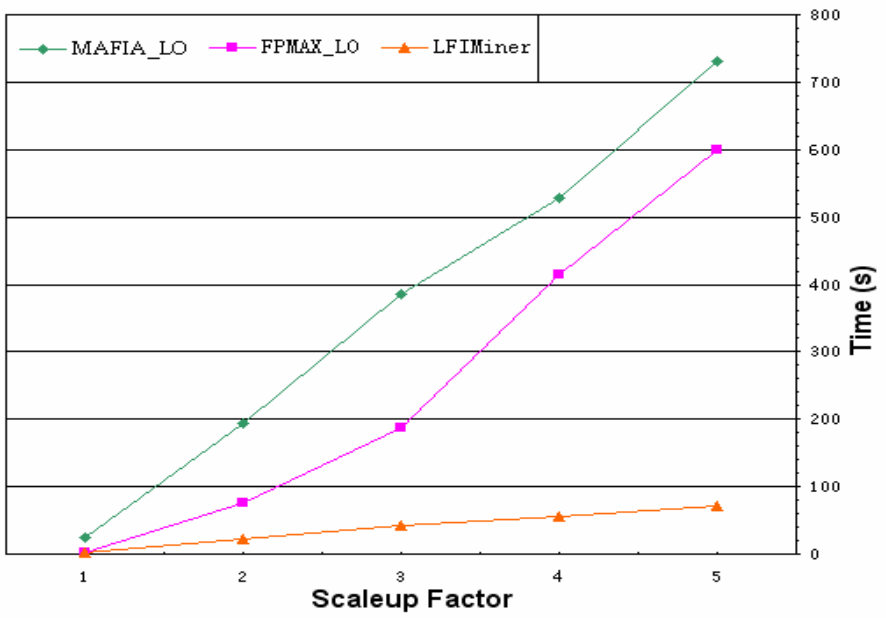


Figure 4.6: Scaleup on *Connect4*

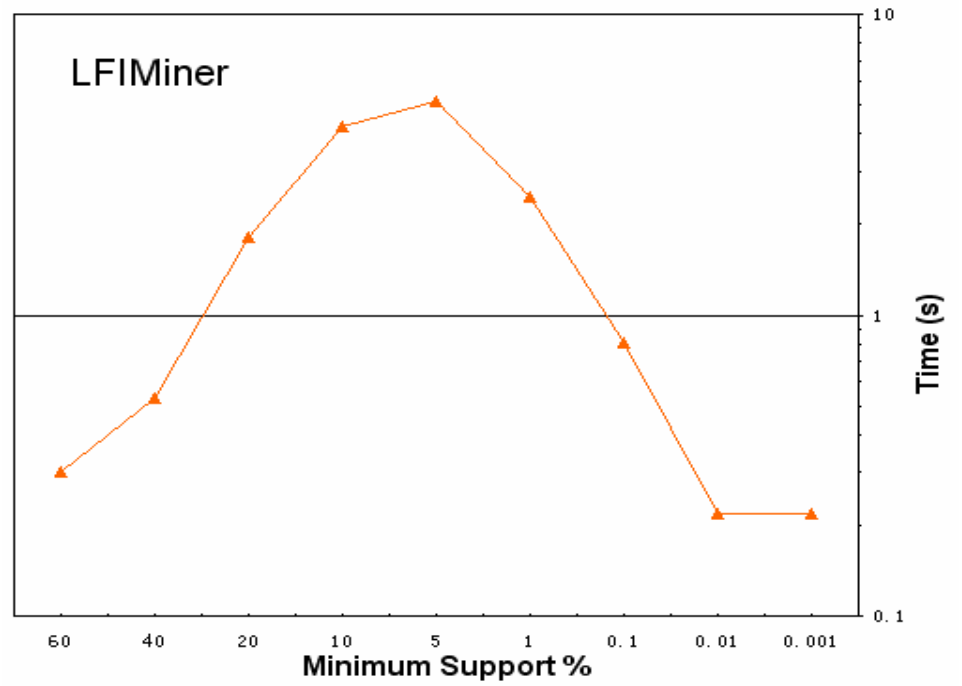


Figure 4.7 (a): Time of LFIMiner on *Chess*

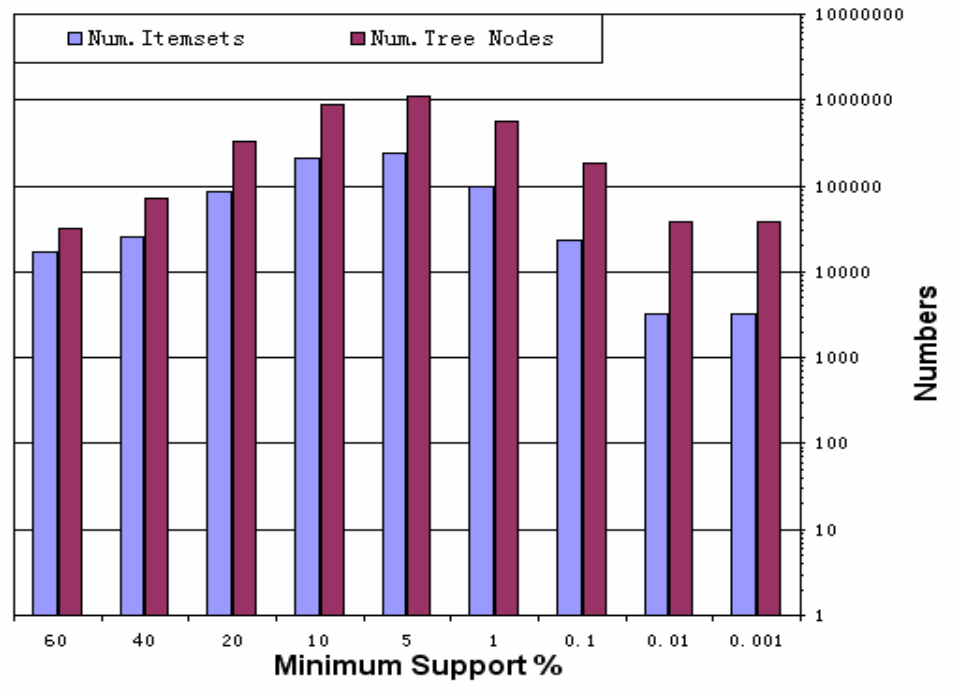


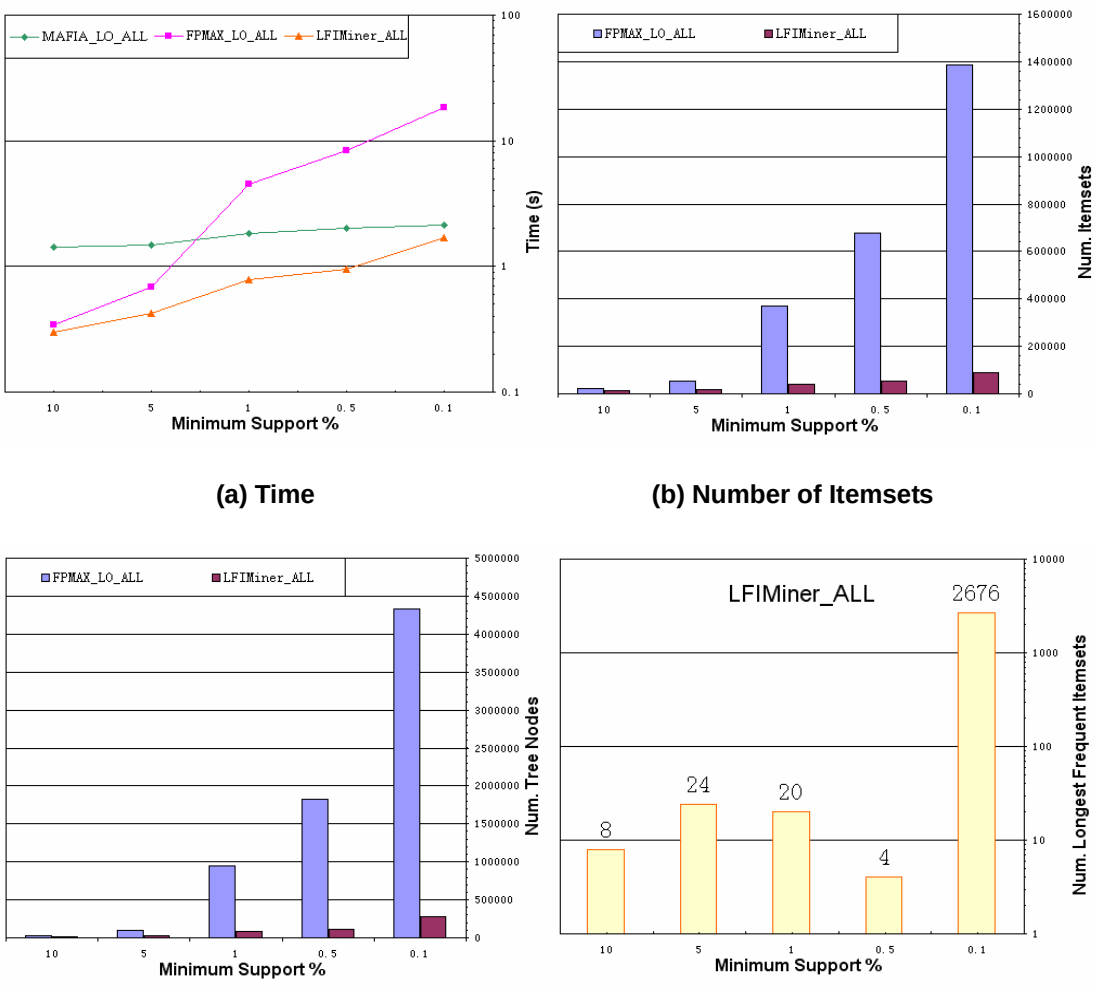
Figure 4.7 (b): Num. Itemsets and Tree Nodes of LFIMiner on *Chess*

There is something interesting here which deserves our attention. Figure 4.7 (a) shows the running time of LFIMiner on *Chess* with different levels of support. As the support decreases, the time does not increase monotonically; instead it first increases, then decreases, and finally keeps steady. Figure 4.7 (b) reflects the variation of number of itemsets processed and tree nodes created by LFIMiner which, we can see, is consistent with the time variation in Figure 4.7 (a). Apparently, as the support decreases, the number of candidate itemsets increases, but at the same time the frequent itemset discovered grows longer, which allows more candidate itemsets to be trimmed. In the first phase, the speed of candidate itemset generation exceeds that of candidate itemset reduction, so the running time increases. In the second phase, the speed of candidate itemset reduction exceeds that of candidate itemset generation, so the running time decreases. In the final phase, in fact, the absolute support reduces to 1, i.e. every transaction is a frequent itemset. In this extreme case, no candidate itemset is generated and thus the running time keeps steady. Similar results were found for *Mushroom, Connect4* and *Pumsb** as well.

4.4 Finding All Longest Frequent Itemsets

Here we compare the results of LFIMiner_ALL with MAFIA_LO_ALL and FPMAX_LO_ALL for finding all the longest frequent itemsets in Figure 4.8-4.11. In each figure, part (a) shows the running time, part (b) shows the number of itemsets processed by FPMAX_LO_ALL and LFIMiner_ALL, part (c) shows the number of FP-tree nodes created by FPMAX_LO_ALL and LFIMiner_ALL, and part (d) shows the number of longest frequent itemsets found. Compared with Figure 4.2-4.5, similar results were found as before, although the time required for mining is longer. From part (d), we

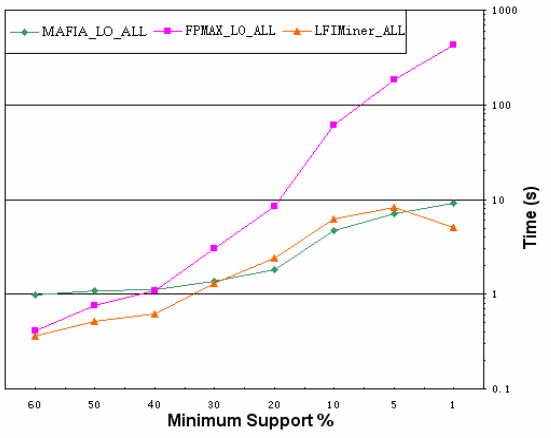
can see in general the number of longest frequent itemsets is under several hundred, which is orders of magnitude smaller than the number of maximal frequent itemsets. For example, at support 10%, the number of maximal frequent itemsets is 547 in *Mushroom*, 2,339,525 in *Chess*, 130,986 in *Connect4*, and 16,437 in *Pumsb**, while the number of longest frequent itemsets is 8, 65, 2 and 1 respectively. Figure 4.12 demonstrates the scalability of the three algorithms, which is similar as before.



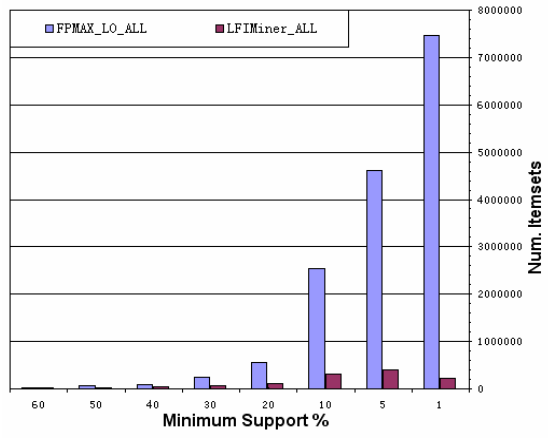
(a) Time **(b) Number of Itemsets**

(c) Number of Tree Nodes **(d) Number of Longest Frequent Itemsets**

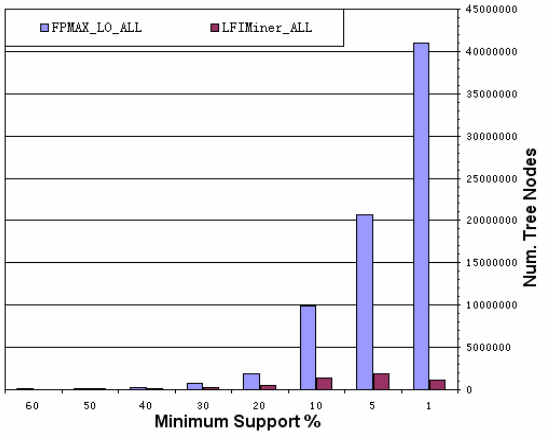
Figure 4.8: Comparison on *Mushroom*



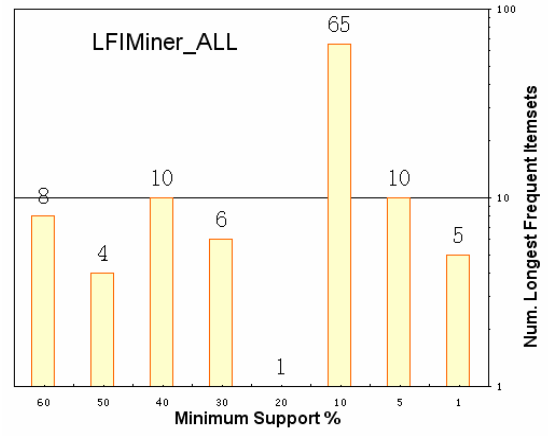
(a) Time



(b) Number of Itemsets

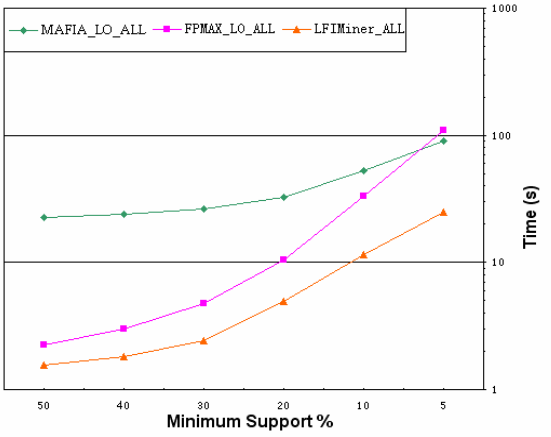


(c) Number of Tree Nodes

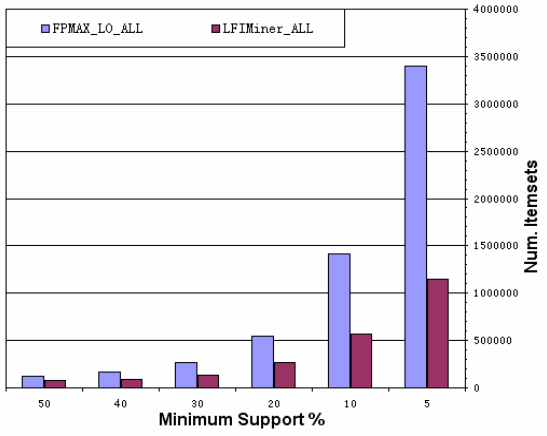


(d) Number of Longest Frequent Itemsets

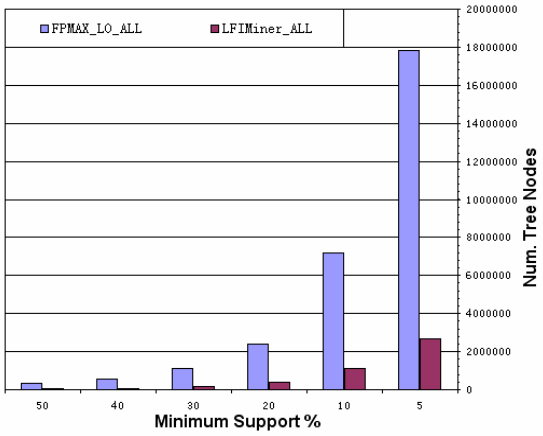
Figure 4.9: Comparison on *Chess*



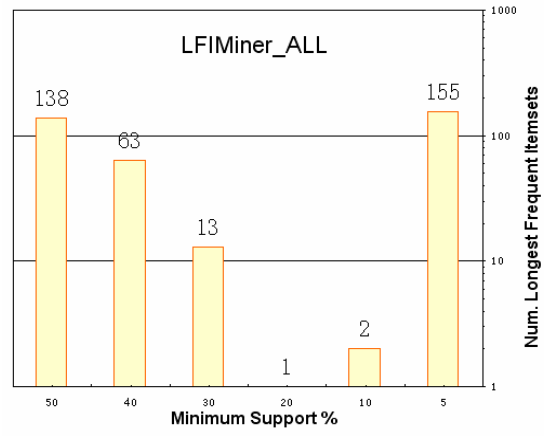
(a) Time



(b) Number of Itemsets

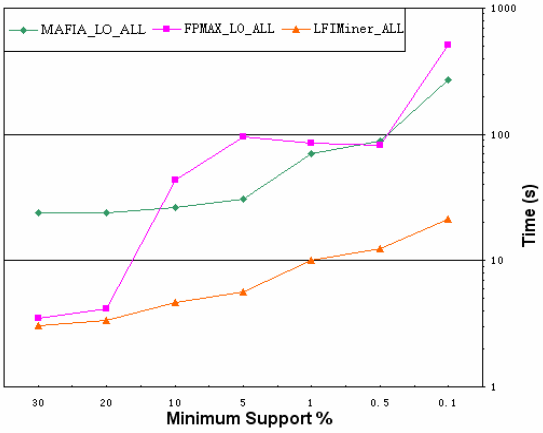


(c) Number of Tree Nodes

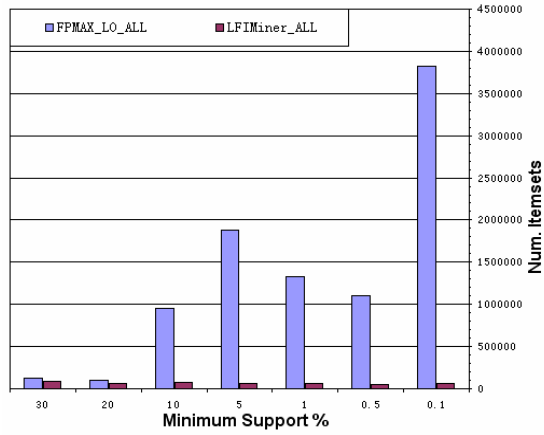


(d) Number of Longest Frequent Itemsets

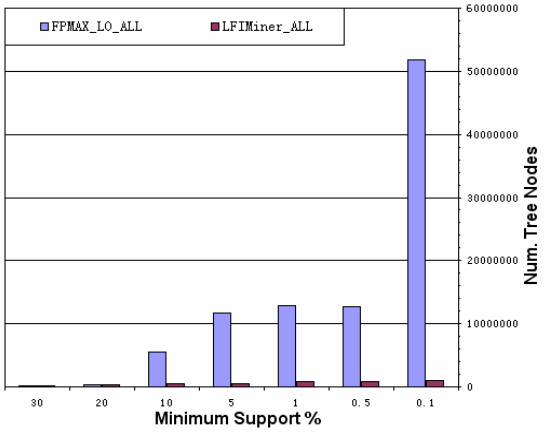
Figure 4.10: Comparison on *Connect4*



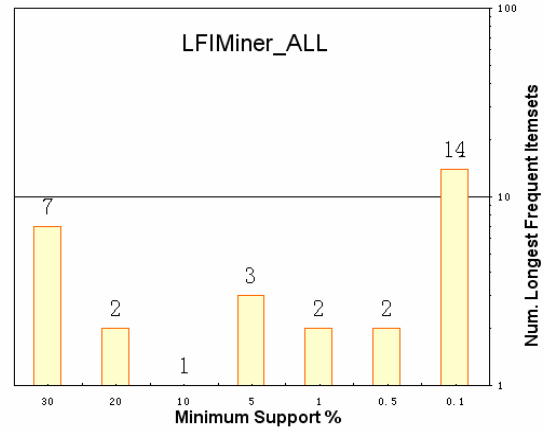
(a) Time



(b) Number of Itemsets



(c) Number of Tree Nodes



(d) Number of Longest Frequent Itemsets

Figure 4.11: Comparison on *Pumsb**

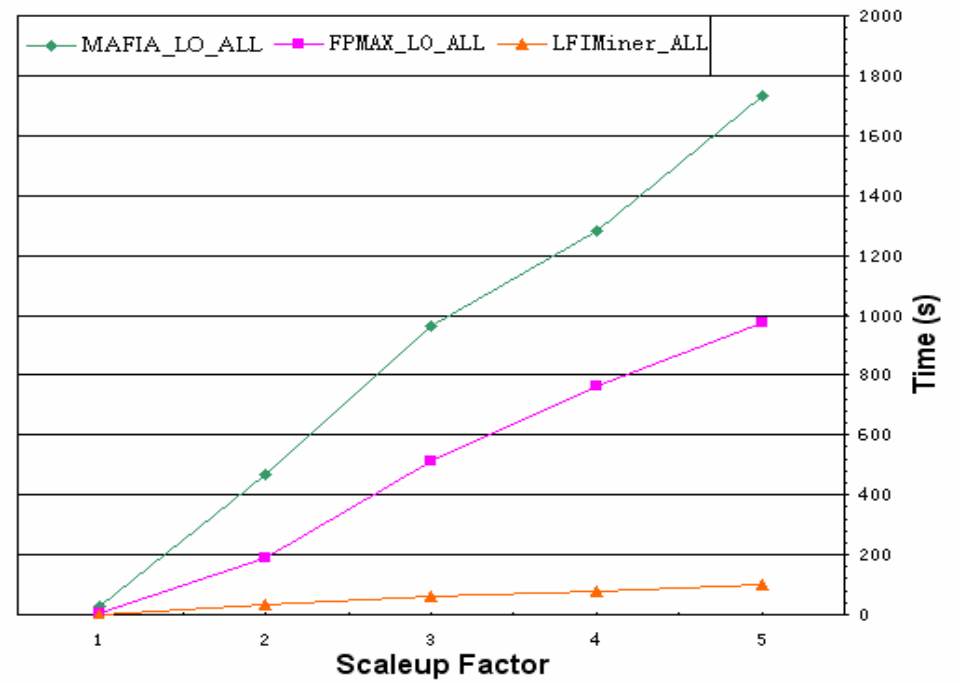


Figure 4.12: Scaleup on *Connect4*

Chapter 5

Using LFI in Clustering

In this chapter, we describe how to explore **LFI** in transaction clustering. First, we introduce our approach which applies **LFI** for clustering. Then we describe the experimental results on some real datasets by comparing our approach with some existing algorithms.

5.1 Algorithm Description

A frequent itemset represents something common to many transactions in a cluster. Therefore, it is a natural way to use frequent itemsets for clustering. [BEX02] [FWE03] apply frequent itemsets into document clustering. In their strategies, documents covering the same frequent itemset are put into the same cluster. Note that **LFI** is the kind of frequent itemsets with maximum length, and intuitively transactions sharing more items have a larger likelihood of belonging to the same cluster. Therefore, it is reasonable to use **LFI** for transaction clustering.

In transaction clustering, the difficulty arises when clusters have overlap. The approach in [EGVB98] does not handle the overlapping case. All clusters found are non-overlapping. The approach in [WXL99] discourages splitting the large items between clusters by increasing inter-cluster cost for overlapping large items. This adversely forces

transactions containing the same large items stick together, and therefore cannot nicely deal with the overlapping case either. The use of longest frequent itemsets can separate long itemsets from short itemsets, therefore solve the overlapping problem among clusters. For example, let's consider the customer buying behavior in one bookstore. From the transaction database, we know some customers only bought Data Mining books and some customers bought Statistics books. At the same time, there are a large number of customers who bought both Data Mining and Statistics books. It is natural to divide the customers into three clusters. The use of longest frequent itemsets can easily separate the customers who bought both Data Mining and Statistics books into one distinct cluster, while this is hard for other traditional transaction clustering methods.

Our approach based on **LFI** for clustering transactions is described in Figure 5.1.

Explanation of the detailed steps of the algorithm

Partition Phase: The key point of our approach for transaction clustering is to make use of the longest frequent itemset which satisfies the minimum support threshold – *min_sup* (it is a percentage number, the absolute support is *min_sup* * |*D*|, where |*D*| is the number of transactions in database *D*). All transactions covering the longest frequent itemset can be grouped together as one cluster. Our method is an iterative process. In each iteration, a cluster is picked to partition. Intuitively, if there are more frequent items in a cluster, the transactions in the cluster are correlated over more items and thus they are more similar. So we pick the cluster with the minimum number of frequent items. This time we use a different threshold *min_sup_item* to determine whether an item is frequent. In the first iteration, the original dataset *D* is partitioned. Step (2) generates a longest frequent

Input:

D: a set of *N* transactions $\{t_1 \ldots t_N\}$

min_sup (percentage number): a user-specified minimum support threshold for finding the longest frequent itemset

min_sup_item (percentage number): a user-specified minimum support threshold for identifying the frequent items

Output:

transaction clusters

Method:

/*Partition Phase*/

(1) Pick the cluster *C*, which contains the minimum number of frequent items among all clusters, (an item *i* in *C* is frequent if supp(*i*) $\geqq$ *min_sup_item* * |*C*|, |*C*| is the number of transactions in *C*), to partition, initially *C* = *D*

(2) Find a longest frequent itemset of *C* having support $\geqq$ *min_sup* * |*C*|

(3) Create a cluster for all transactions in *C* that cover the longest frequent itemset

(4) Create a cluster for remaining transactions in *C*

(5) Repeat steps (1) - (4) until the desired number of clusters is reached or other user-specified stop condition is satisfied

/*End of Partition Phase*/

/*Refinement Phase*/

(1) Move each transaction into the "closest" cluster. The "closeness" between a transaction and a cluster is measured as follows: $Close(t, C) = E_t \cap E_C$, where E_t is the set of items of transaction *t*, and E_C is the set of frequent items of cluster *C*, (an item *i* in *C* is frequent if supp(*i*) $\geqq$ *min_sup_item* * |*C*|, |*C*| is the number of transactions in *C*)

(2) Repeat step (1) until no transaction moves

/*End of Refinement Phase*/

Figure 5.1: The Clustering Approach Using LFI

itemset using the LFIMiner algorithm. In step (3), all transactions that cover the longest frequent itemset are put into a cluster. In step (4), the remaining transactions are put into another cluster. In general, this cluster has a larger chance to be further partitioned in the next iteration than the cluster formed by the longest frequent itemset. The termination condition we can use in the practice consists of stopping when the desired number of clusters is reached or the number of frequent items in the clusters is above a user-specified value.

Refinement Phase: There may have some transactions which are "closer" to other clusters than the clusters they currently belong to. For example, in step (4) of the partition phase, the leftovers form a cluster. Considering the cluster formed by the longest frequent itemset in step (3), maybe some leftovers contain a majority of items of the longest itemset, and maybe they are "closer" to this cluster. So there is a need to move these transactions into their "closest" clusters. The "closeness" between a transaction and a cluster is measured as follows:

$$Close(t, C) = E_t \cap E_C$$

where E_t is the set of items of transaction t, and E_C is the set of frequent items of cluster C. Refinement is also iterative; it stops until no transaction moves in one single iteration.

5.2 Experimental Results

We test our algorithm on the *Mushroom*, *Congressional Votes*, *Zoo* and *Soybean-small* datasets, in which *Mushroom* and *Congressional Votes* are used in ROCK [GRS99], OAK [XD01] and LargeItem [WXL99] [YCC02], *Zoo* is used in CoFD [ST02].

The *Mushroom* dataset is from *The Audubon Society Field Guide to North American Mushrooms*. Each record contains information that describes the physical characteristics of a single mushroom. There are 8,124 records, 4,208 are categorized as Edible and 3,196 as Poisonous. By treating the value of each attributes as items of transactions, we converted all the 22 categorical attributes to transactions with 117 distinct items (distinct attribute values). The *Congressional Votes* dataset is the United States Congressional Votes Records in 1984. Each record corresponds to one congressman's votes on 16 issues. (A few records contain missing values, which we treat as NO). There are 435 records, 168 for Republicans and 267 for Democrats. The *Zoo* dataset contains 17 Categorical-valued attributes, which describes the physical characteristics of a single animal. There are 101 records, which are categorized into 7 classes. The *Soybean-small* dataset contains 35 Categorical-valued attributes, which describes the physical characteristics of a single soybean. There are 47 records belonging to 4 different classes. All these datasets are from the UCI Machine Learning Repository [UCIMLR].

Results on *Mushroom*

We test our algorithm at different levels of *min_sup* and *min_sup_item* thresholds. We try different *min_sup* and *min_sup_item* values from 30% to 50%, with a step of 2%. We set the number of clusters to be 21, which is same as in ROCK and OAK. The results are shown in Figure 5.2, Figure 5.3 and Figure 5.4. In Figure 5.2, the x-axis is *min_sup*, and the colorful columns from pearl blue to yellow represent different levels of *min_sup_item* from 30% to 50%. While in Figure 5.3, the x-axis is *min_sup_item*, and the colorful columns from pearl blue to yellow represent different levels of *min_sup* from 30% to 50%. The y-axis in both figures is the purity metric, which is computed by

summing up the larger one of the number of edibles and the number of poisonous in every cluster. It has a maximum of 8,124. Figure 5.4 shows the running time. The x-axis is *min_sup*, the y-axis is the running time in seconds. The blue line represents the average algorithm time when *min_sup_item* is from 30% to 50%, while the pink line represents the average time for mining LFI (in the partition phase) when *min_sup_item* is from 30% to 50%. For example, when *min_sup* is 30%, the average algorithm time is 2.83s, and the average time for mining LFI is 1.55s.

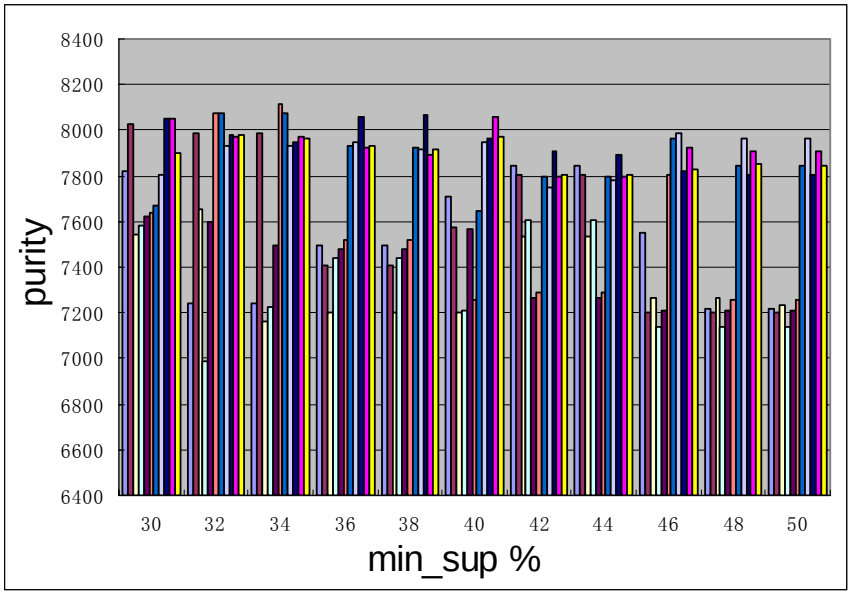


Figure 5.2: The results at different levels of min_sup on *Mushroom*

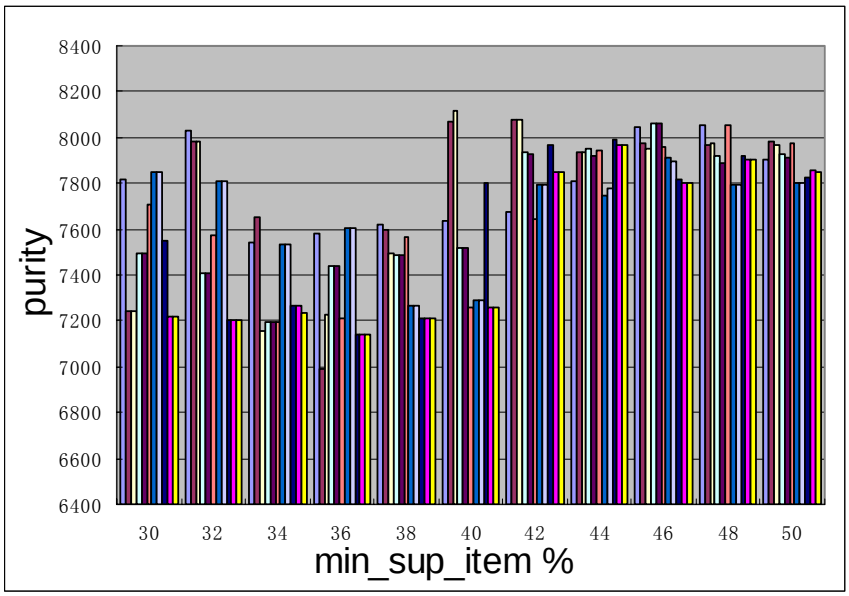


Figure 5.3: The results at different levels of min_sup_item on *Mushroom*

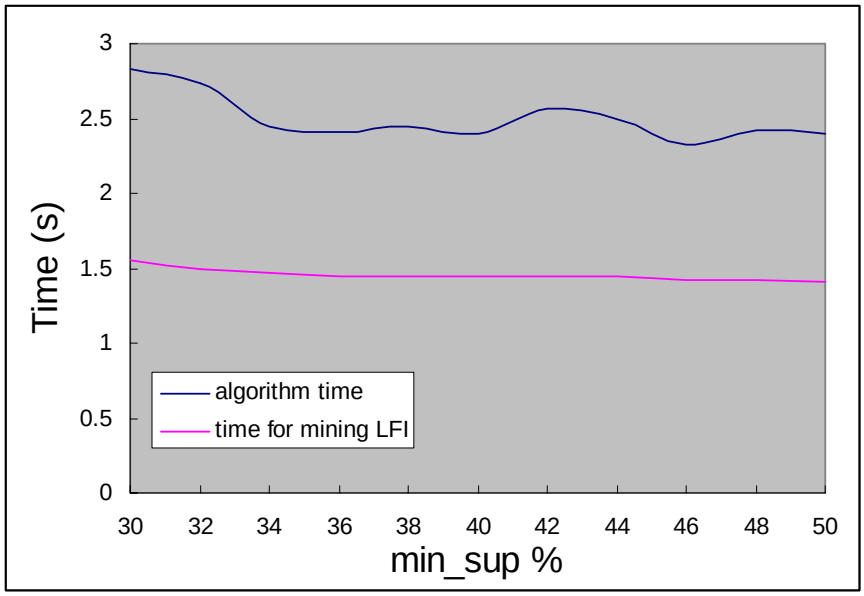


Figure 5.4: Running time at different levels of min_sup on *Mushroom*

From Figure 5.2 and Figure 5.3, we can see, in general, for all levels of *min_sup* (30%-50%), the purity for higher levels of *min_sup_item* (40%-50%) is larger than that

for lower levels of *min_sup_item* (30%-40%). This indicates setting *min_sup_item* between 40% and 50% will achieve a better clustering result. Figure 5.4 shows that as *min_sup* increases, the time for mining LFI decreases, while the algorithm time fluctuates slightly but irregularly. It is because normally mining LFI is faster at higher levels of *min_sup* than that at lower levels of *min_sup*, however, this is not definite for the whole algorithm execution because the algorithm time includes the time in the refinement phase. As *min_sup* increases, the time for mining LFI goes down, while it is possible that the time for refinement goes up. Similar results are found in the other three datasets (please refer to Figure 5.7, Figure 5.10 and Figure 5.13). We compare the clustering results of different algorithms as follows:

LargeItem					
Cluster No	Num. Edible	Num. Poisonous	Cluster No	Num. Edible	Num. Poisonous
1	94	0	8	0	287
2	13	0	9	61	3,388
3	6	0	10	372	77
4	682	26	11	9	0
5	2,631	30	12	19	10
6	121	37	13	21	0
7	69	61	14	110	0
ROCK (OAK)					
Cluster No	Num. Edible	Num. Poisonous	Cluster No	Num. Edible	Num. Poisonous
1	96	0	12	48	0
2	0	256	13	0	288
3	704	0	14	192	0
4	96	0	15	32	72
5	768	0	16	0	1,728
6	0	192	17	288	0
7	1,728	0	18	0	8
8	0	32	19	192	0
9	0	1,296	20	16	0
10	0	8	21	0	36
11	48	0			

Our algorithm (21 clusters)					
Cluster No	Num. Edible	Num. Poisonous	Cluster No	Num. Edible	Num. Poisonous
1	1,726	0	12	48	6
2	0	1,728	13	48	0
3	0	1,296	14	0	36
4	0	192	15	0	40
5	380	0	16	512	0
6	144	0	17	0	256
7	0	216	18	96	0
8	768	0	19	0	72
9	192	0	20	0	72
10	144	0	21	54	2
11	96	0			
Our algorithm (24 clusters)					
Cluster No	Num. Edible	Num. Poisonous	Cluster No	Num. Edible	Num. Poisonous
1	1,728	0	13	0	256
2	0	1,728	14	192	0
3	0	1,296	15	288	0
4	192	0	16	96	0
5	192	0	17	0	72
6	0	144	18	0	72
7	48	0	19	96	0
8	48	0	20	0	72
9	0	36	21	0	40
10	512	0	22	16	0
11	0	192	23	32	0
12	768	0	24	0	8

Table 5.1: Clustering Results on *Mushroom*

We present two clustering results of our algorithm, one is 21 clusters (*min_sup* = 34%, *min_sup_item* = 40%), the other is 24 clusters (*min_sup* = 40%, *min_sup_item* = 47%). From Table 5.1, LargeItem produces many impure clusters. ROCK, OAK (the clusters produced by ROCK and OAK are identical) and our algorithm achieve similar results. ROCK and OAK produce 21 clusters, among which one is impure (72 poisonous and 32 edibles, purity = 8,092). If our algorithm produces 21 clusters, two of them are

impure (one is 6 poisonous and 48 edibles, the other is 2 poisonous and 54 edibles, purity = 8,116). If our algorithm produces 24 clusters, all are pure (purity = 8,124).

Results on *Congressional Votes*

Similar experiments are done on the *Congressional Votes* dataset. We try different *min_sup* and *min_sup_item* values from 30% to 60%, with a step of 3%. We set the number of clusters to be 3, which is same as in ROCK and OAK (outliers are considered as being in an outlier cluster). The results are shown in Figure 5.5, Figure 5.6 and Figure 5.7. The maximum purity is 435.

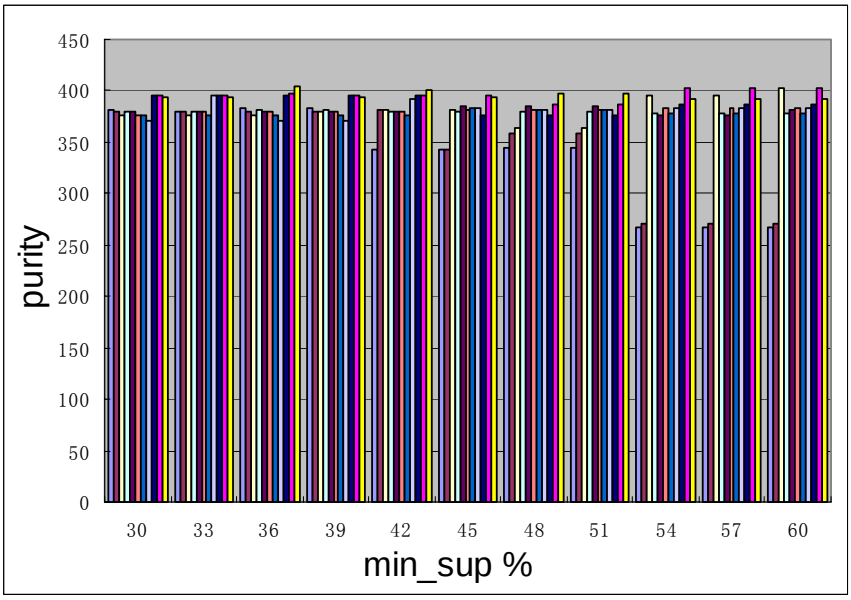


Figure 5.5: The results at different levels of min_sup on *Congress*

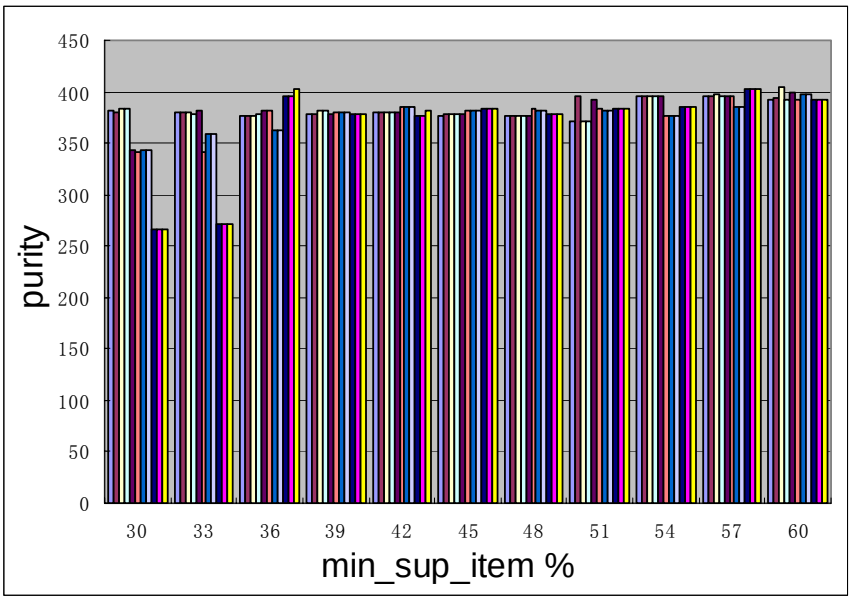


Figure 5.6: The results at different levels of min_sup_item on *Congress*

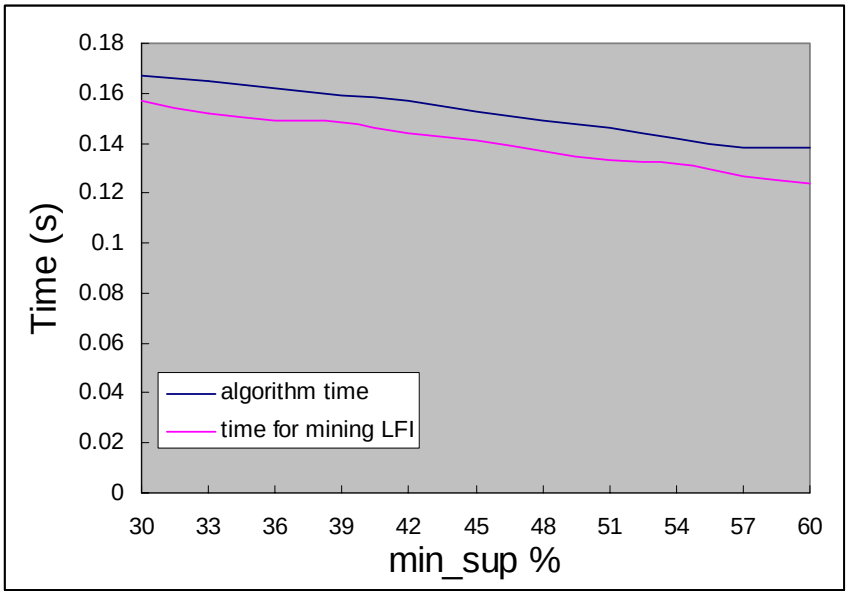


Figure 5.7: Running time at different levels of min_sup on *Congress*

From Figure 5.5 and Figure 5.6, we can see, in general, for all levels of *min_sup* (30%-60%), the purity for higher levels of *min_sup_item* (54%-60%) is larger than that

for lower levels of *min_sup_item* (30%-54%). The higher purity achieved by setting *min_sup_item* to a relatively high value (54%-60%) reflects the fact that democrats and republican do vote similarly on many issues. Figure 5.7 shows that as *min_sup* increases, both the algorithm time and the time for mining LFI decrease. We also notice that the time for mining LFI is close to the algorithm time. It won't take much time in the refinement phase if the dataset is small. The clustering results are presented in Table 5.2.

	LargeItem (Basic in [YCC02])			**ROCK**		
Cluster No	Num. Rep	Num. Demo	Purity	Num. Rep	Num. Demo	Purity
1	125	46	73.1%	144	22	86.7%
2	43	221	83.7%	5	201	97.6%
Outliers	NA	NA	NA	19	44	69.8%
	OAK			**Our algorithm**		
Cluster No	Num. Rep	Num. Demo	Purity	Num. Rep	Num. Demo	Purity
1	147	22	87.0%	152	15	91.0%
2	5	201	97.6%	10	204	95.3%
Outliers	16	44	73.3%	6	48	88.9%

Table 5.2: Clustering Results on *Congressional Votes*

We set *min_sup* to be 36% and *min_sup_item* to be 60% in our algorithm. From Table 5.2, all algorithms produce two big clusters. As measured by the class purity of each cluster, cluster 1 produced by our algorithm is better, while cluster 2 produced by ROCK and OAK is better. If we treat the rest as an outlier cluster, the cluster produced by our algorithm also has high class purity. The purity in the two big clusters (clusters 1 & 2) of our algorithm is 356/381 (93.4%), while it is 345/372 (92.7%) in ROCK and 348/375 (92.8%) in OAK respectively. If we consider the purity in all the three clusters (including the outlier cluster), the purity produced by our algorithm is 92.9%, while it is 89.4% in ROCK and 90.1% in OAK. We can conclude our algorithm produces slightly better result

on *Congressional Votes* than ROCK and OAK.

Results on *Zoo*

We try different *min_sup* and *min_sup_item* values from 20% to 60%, with a step of 4%. We set the number of clusters to be 7, which is the number of animal classes in the dataset. The results are shown in Figure 5.8, Figure 5.9 and Figure 5.10. The maximum purity is 101.

From Figure 5.8 & 5.9, we can see, in general, for all levels of *min_sup* (20%-60%), the purity for higher levels of *min_sup_item* (52%-60%) is larger than that for lower levels of *min_sup_item* (20%-52%). Figure 5.10 shows the similar results as before. We compare the clustering result of our algorithm with CoFD in Table 5.3.

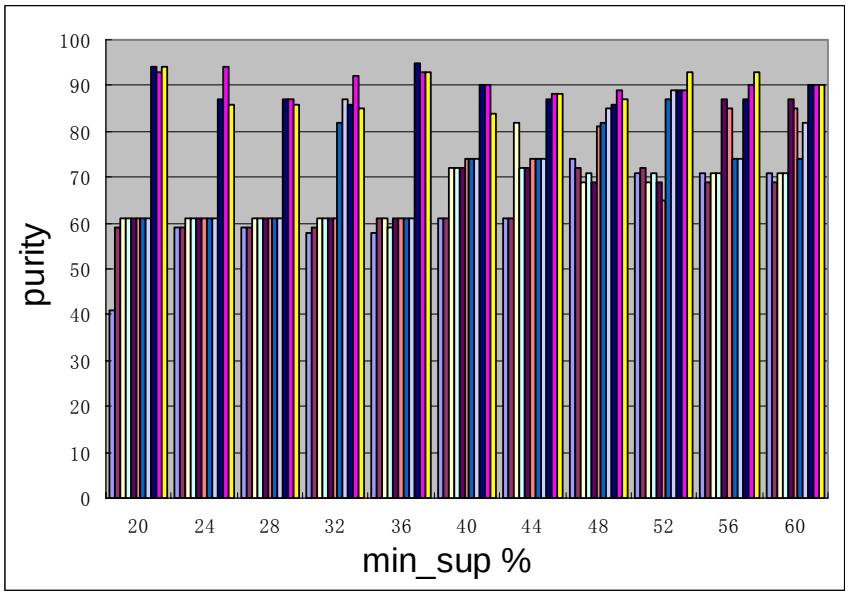


Figure 5.8: The results at different levels of min_sup on *Zoo*

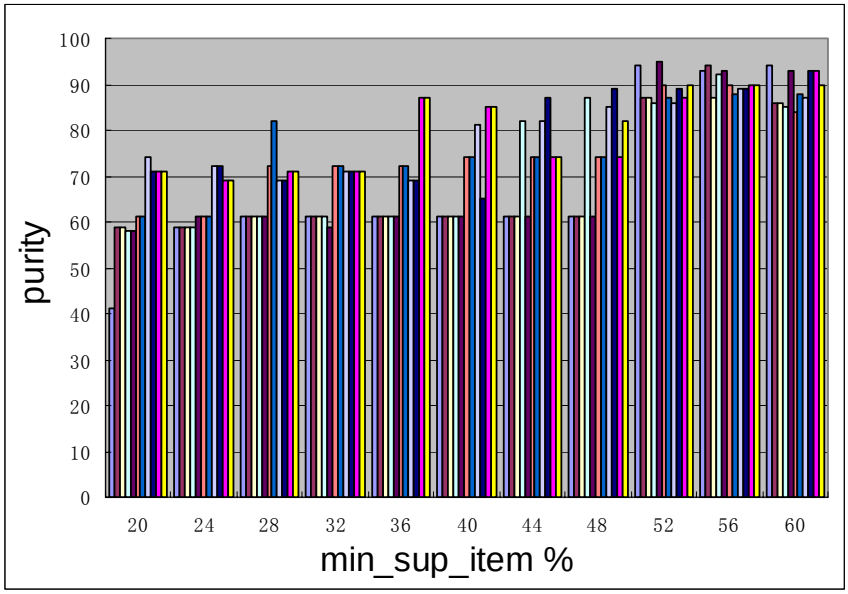


Figure 5.9: The results at different levels of min_sup_item on *Zoo*

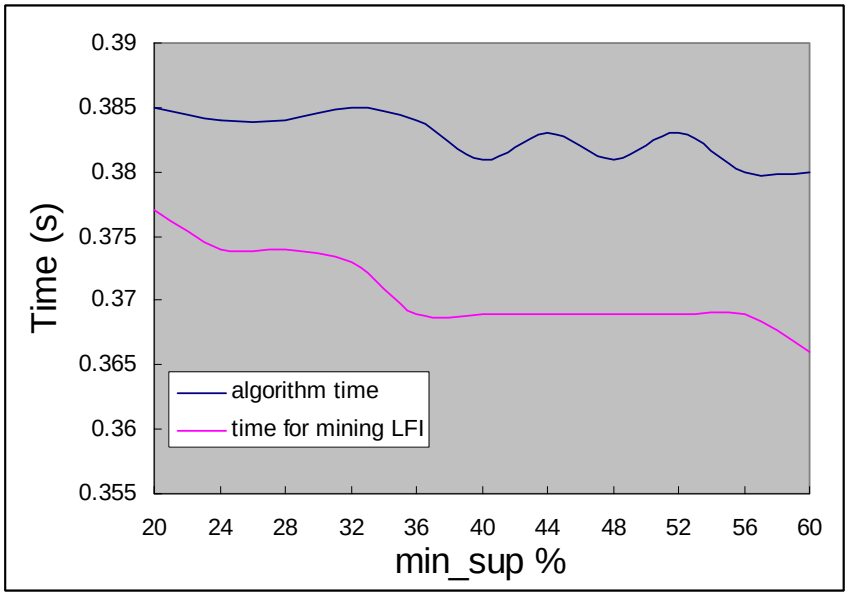


Figure 5.10: Running time at different levels of min_sup on *Zoo*

CoFD							
Cluster No	Num. Class #1	Num. Class #2	Num. Class #3	Num. Class #4	Num. Class #5	Num. Class #6	Num. Class #7
1	39	0	0	0	0	0	0
2	0	20	0	0	0	0	0
3	2	0	1	13	0	0	4
4	0	0	0	0	0	0	1
5	0	0	2	0	0	0	0
6	0	0	0	0	0	8	5
7	0	0	2	0	3	0	0
Our algorithm							
Cluster No	Num. Class #1	Num. Class #2	Num. Class #3	Num. Class #4	Num. Class #5	Num. Class #6	Num. Class #7
1	34	0	0	0	0	0	0
2	0	20	0	0	0	0	0
3	0	0	1	13	0	0	0
4	0	0	0	0	0	0	8
5	7	0	0	0	0	0	0
6	0	0	0	0	0	8	2
7	0	0	4	0	4	0	0

Table 5.3: Clustering Results on *Zoo*

There are only 100 records in the dataset used in CoFD, which contains 3 records of Class #5, while our 101-record dataset contains 4 records of Class #5. We get the result by setting *min_sup* to be 20% and *min_sup_item* to be 60%. From Table 5.3, the two algorithms achieve similar results, however, the cluster 3 in our algorithm is better than that in CoFD, and our algorithm aggregates 8 of 10 records of Class #7 into a single cluster (cluster 4) while these records are distributed into 3 different clusters in CoFD. The purity in our algorithm is 94/101 (93%), while it is 86/100 (86%) in CoFD. Thus we can conclude our algorithm produces better result on *Zoo* than CoFD.

Results on *Soybean-small*

We try different *min_sup* and *min_sup_item* values from 30% to 60%, with a step of

3%. We set the number of clusters to be 4, which is the number of soybean classes in the dataset. The results are shown in Figure 5.11, Figure 5.12 and Figure 5.13. The maximum purity is 47.

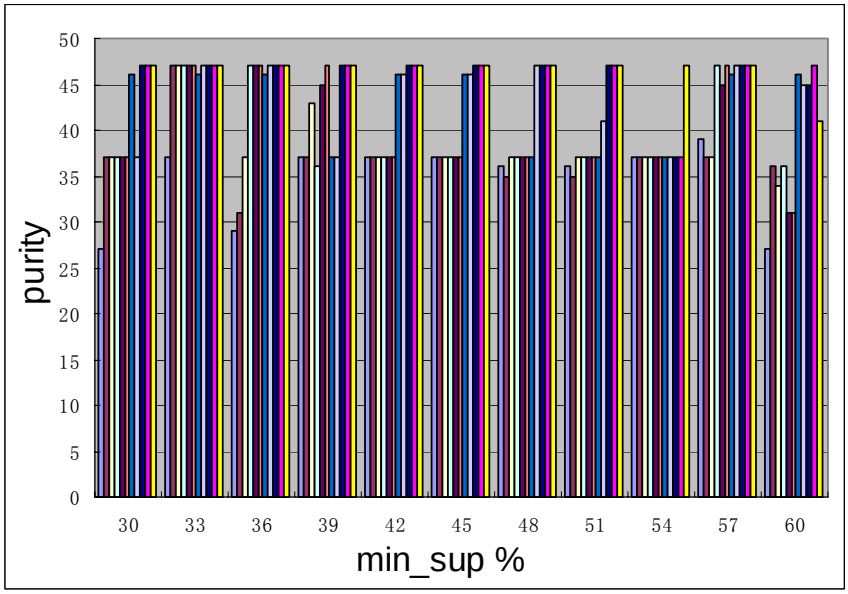


Figure 5.11: The results at different levels of min_sup on *Soybean-small*

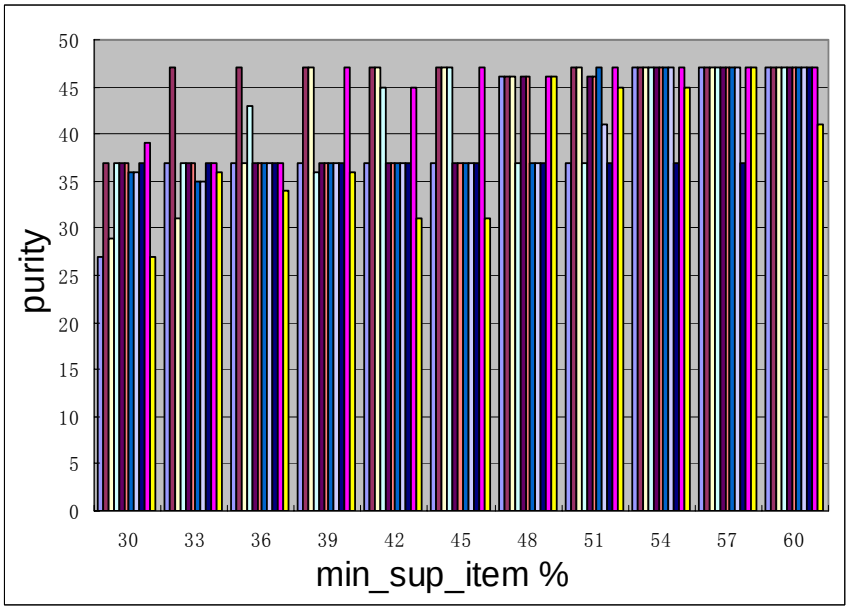


Figure 5.12: The results at different levels of min_sup_item on *Soybean-small*

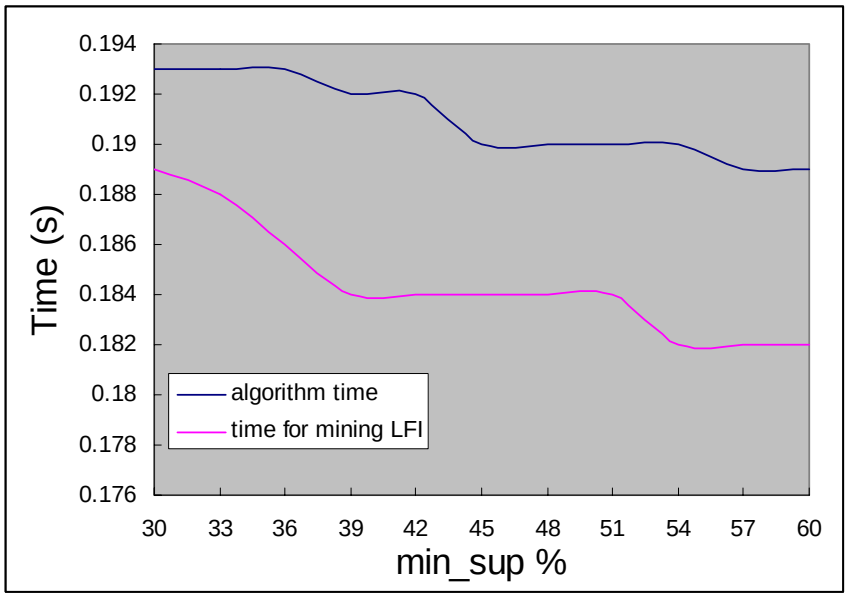


Figure 5.13: Running time at different levels of min_sup on *Soybean-small*

From Figure 5.11 & 5.12, in general, as before, for all levels of *min_sup* (30%-60%), the purity for higher levels of *min_sup_item* (48%-60%) is larger than that for lower levels of *min_sup_item* (30%-48%). The results in Figure 5.13 are similar as before. The clustering results are presented in Table 5.4.

Our algorithm				
Cluster No	Num. Class #1	Num. Class #2	Num. Class #3	Num. Class #4
1	0	0	0	17
2	0	10	0	0
3	0	0	10	0
4	10	0	0	0

Table 5.4: Clustering Results on *Soybean-small*

In Table 5.4, by setting *min_sup* to be 30% and *min_sup_item* to be 60%, our algorithm successfully divides the dataset into 4 pure clusters (purity = 47).

From the experiments of the above four datasets, we find setting *min_sup_item* to a relatively high value (50%-60%) will normally achieve a better clustering result. The probable reason is that all the four datasets are dense datasets. For some sparse datasets, maybe *min_sup_item* should be set to be a low value to achieve good clustering.

To test the scalability of our algorithm, we ran it on the *Mushroom* dataset, which was vertically enlarged by duplicating the transactions. The x-axis is the scaleup factor. *min_sup* is fixed at 34%, *min_sup_item* is fixed at 40%, and the number of clusters is set to be 21. The results are shown in Figure 5.14. We can see our algorithm scales almost linearly with database size.

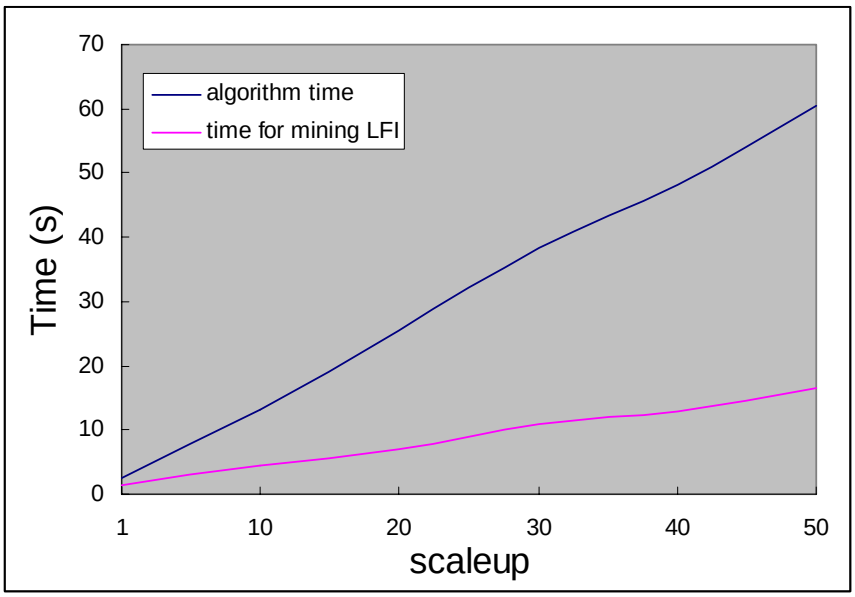


Figure 5.14: Scaleup of Our Algorithm on *Mushroom*

5.3 Conclusions

Our research on clustering transactions using **LFI** is preliminary. Although we don't expect our approach will achieve good results for all types of datasets, the good results on the above 4 datasets indicate that **LFI** is a promising technique to be used in clustering.

Chapter 6

Conclusions and Future Work

In this thesis, we introduced the problem of finding the longest frequent itemset, and we proposed an efficient algorithm called LFIMiner, which is based on the conceptions of the FP-tree structure and pattern fragment growth, to fulfill our task. We also found some real world applications where this problem needs to be solved. The FP-tree structure stores compressed, crucial information about frequent itemsets in a database and, meanwhile, the pattern growth method avoids the costly candidate set generation and test. Further, we integrate several techniques in our algorithm. The CPP and FIP pruning schemes help LFIMiner reduce the search space dramatically by removing noncontributing conditional itemsets and in turn narrowing the conditional FP-trees. Dynamic Reordering reduces the size of an FP-tree by keeping more frequent items closer to the root in the FP-tree.

In comparison with the MAFIA_LO and FPMAX_LO algorithms, the LFIMiner algorithm is a faster method of mining the longest frequent itemset. A detailed analysis of each component showed that the frequent item pruning (FIP) is quite effective in reducing the search space because of its recursive process. LFIMiner also exhibits a good scalability. In addition, a variant of this algorithm is efficient for mining all the longest frequent itemsets after some small modifications.

In Chapter 5, we propose an approach which uses **LFI** in clustering. The good clustering results on some real datasets exhibit its potential to be used for clustering. Thus an interesting research issue is to further exploit **LFI** in clustering. Other interesting future work includes finding a faster method of mining **LFI**, investigating more use of **LFI**, finding other interesting itemsets with the FP-tree structure, and so on.

Bibliography

[AAP00] R. Agarwal, C. Aggarwal and V. V. V. Prasad. Depth First Generation of Long Patterns. In *Proc. ACM SIGMOD Conference*, 2000, Boston, Massachusetts, United States, pages 108-118.

[AIS93] R. Agrawal, T. Imielinski and A. N. Swami. Mining Association Rules between Sets of Items in Large Databases. In *Proc. ACM SIGMOD'93*, Washington, DC, pages 207-216.

[AS94] R. Agrawal and R. Srikant. Fast Algorithm for Mining Association Rules. In *Proc. VLDB'94*, Santiago, Chile, pages 487-499.

[B98] R. J. Bayardo. Efficiently Mining Long Patterns from Databases. In *Proc. ACM SIGMOD'98*, 1998, Seattle, Washington, pages 85-93.

[BCG01] D. Burdick, M. Calimlim and J. Gehrke. MAFIA: A Maximal Frequent Itemset Algorithm for Transaction Databases. In *Proc. ICDE'01*, 2001, Washington, DC, pages 443-452.

[BEX02] F. Beil, M. Ester, and X. Xu. Frequent Term-Based Text Clustering. In *Proc. 8th Int. Conf. on Knowledge Discovery and Data Mining (KDD)'2002*, Edmonton, Alberta, Canada, 2002, pages 436-442.

[BMUT97] S. Brin, R. Motwani, J. Ullman and S. Tsur. Dynamic Itemset Counting and Implication Rules for Market Basket Data. In *Proc. ACM-SIGMOD'97 on Management of Data*, Tucson, Arizona, United States, pages 255-264.

[CZ03] W. Cheung and O. R. Zaiane. Incremental mining of frequent patterns without candidate generation or support constraint. In *Proceedings of the Seventh International Database Engineering and Applications Symposium*, 2003, Hong Kong, SAR, pages 111-116.

[EGVB98] Eui-Hong Han, George Karypis, Vipin Kumar, and Bamshad Mobasher: Hypergraph Based Clustering in High-Dimensional Data Sets: A Summary of Results. IEEE Data Eng. Bull. 21(1): 15-22 (1998).

[FWE03] B. C. M. Fung, K. Wang, and M. Easter. Hierarchical Document Clustering Using Frequent Itemsets. In *Proc. SDM'03*, San Francisco, CA, May 1-3, 2003, pages 59-70.

[GRS99] S. Guha, R. Rastogi, and K. Shim. ROCK: A Robust Clustering Algorithm for Categorical Attributes. In *Proc. ICDE. 1999*, March 23-26, 1999, Sydney, Australia, pages 512-521.

[GZ01] K. Gouda and M. J. Zaki. Efficiently Mining Maximal Frequent Itemsets. In *Proc. 1st IEEE Int'l Conf. on Data Mining*, San Jose, Nov. 2001, pages 163-170.

[GZ03] G. Grahne and J. Zhu. High Performance Mining of Maximal Frequent Itemsets. In *Proceeding of the sixth SIAM International Workshop on High Performance Data Mining (HPDM '03)*, San Francisco, CA, May 2003.

[HGN00] J. Hipp, U. Guntzer and G. Nakaeizadeh. Algorithms for Association Rule Mining – A General Survey and Comparison. In *Proc. ACM SIGKDD'00*, 2000, Dallas, USA, pages 58-64.

[HK01] J. Han, M. Kamber. Data Mining: Concepts and Techniques. San Francisco: Morgan Kaufmann Publishers, ISBN 1-55860-489-8, 2001.

[HPY00] J. Han, J. Pei and Y. Yin. Mining Frequent Patterns without Candidate Generation. In *Proc. ACM SIGMOD'00*, 2000, Dallas, Texas, United States, pages 1-12.

[LK98] D. Lin and Z. M. Kedem. Pincer-Search: A New Algorithm for Discovering the Maximum Frequent Itemset. In *Proc. the 6th International Conference on Extending Database Technology (EDBT'98)*, 1998, Valencia, Spain, pages 105-119.

[P01] H. Pinto. Multi-Dimensional Sequential Pattern Mining. *M.S. Thesis*, 2001, Simon Fraser University, Canada.

[PCY95] J. S. Park, M.-S. Chen and P. S. Yu. An Effective Hash Based Algorithm for

Mining Association Rules. In *Proc. ACM SIGMOD'95*, May 1995, San Jose, CA, pages 175-186.

[PHM00] J. Pei, J. Han and R. Mao. CLOSET: An efficient algorithm for mining frequent closed itemsets. In *ACM SIGMOD Workshop on Research Issues in Data Mining and Knowledge Discovery*, 2000, Dallas, USA, pages 21-30.

[PHP01] J. Pei, J. Han, H. Pinto, Q. Chen, U.Dayal and M.-C. Hsu. PrefixSpan: Mining Sequential Patterns Efficiently by Prefix-Projected Pattern Growth. In *Proc. 2001 International Conference on Data Engineering (ICDE'01)*, 2001, Heidelberg, Germany, pages 215-224.

[R92] R. Rymon. Search through Systematic Set Enumeration. In *Proc. the Third International Conference on Principles of Knowledge Representation and Reasoning*, 1992, Cambridge, MA, pages 539-550.

[SA96] R. Srikant and R. Agrawal. Mining Sequential Patterns: Generalizations and Performance Improvements. In *Proc. the Fifth International Conference on Extending Database Technology (EDBT'96)*, 1996, Avignon, France, pages 3-17.

[SHS00] P. Shenoy, J. R. Haritsa, S. Sudarshan, G. Bhalotia, M. Bawa and D. Shah. Turbo-charging Vertical Mining of Large Databases. In *Proc. ACM SIGMOD'00*, 2000, Dallas, Texas, United States, pages 22-33.

[ST02] Shenghuo Zhu and Tao Li. An Algorithm for Non-distance Based Clustering in High Dimensional Spaces, UR-CS-TR763, 2002.

[T96] H. Toivonen. Sampling large databases for association rules. In *Proceeding of International Conference Very Large Data Bases*, 1996, Bombay, India, pages 134-145.

[UCIMLR] UCI Machine Learning Repository. *University of California, Irvine*. http://www.ics.uci.edu/~mlearn/MLRepository.html.

[WXL99] K. Wang, C. Xu, and B. Liu. Clustering Transactions Using Large Items. In *Proc. CIKM'99*, Kansas City, Missouri, United States, pages 483-490.

[WZ02] M. Wojciechowski and M. Zakrzewicz: Dataset Filtering Techniques in Constraint-Based Frequent Pattern Mining. *Pattern Detection and Discovery 2002*, pages 77-91.

[XD01] Y. Xiao and M. H. Dunham. Interactive Clustering for Transaction Data. In *Proceedings of the Third International Conference on Data Warehousing and Knowledge Discovery*, September 05-07, 2001, Munich, Germany, pages121-130.

[YCC02] C.-H. Yun, K.-T. Chuang, and M.-S. Chen. Self-Tuning Clustering: An Adaptive Clustering Method for Transaction Data. In *Proceedings of the*

Fourth International Conference on Data Warehousing and Knowledge Discovery, 2002, Aix-en-Provence, France, pages 42-51.

[Z01] M. J. Zaki. SPADE: An Efficient Algorithm for Mining Frequent Sequences. *Machine Learning Journal*, Vol. 42, pages 31-60, 2001.

[ZG01] M. J. Zaki and K. Gouda. Fast vertical mining using Diffsets. *TR 01-1*, 2001, CS Dept., RPI.

[ZH02] M. J. Zaki and C.-J. Hsiao. CHARM: An Efficient Algorithm for Closed Itemset Mining. In *2nd SIAM International Conference on Data Mining*, April 2002, Arlington, USA, pages 12-28.

Appendix

• The MAFIA_LO Algorithm

Input: *C*: the current node, *IsHUT*: whether this is a HUT check

Global: *lfi*: the longest frequent itemset found so far

Local: *HUT*: the set of head & tail, *NewNode*: a node

PEPSet: a set of items that are moved from tail to head by PEP pruning

Output: The *LFI* that is the longest frequent itemset

Method: Call **MAFIA_LO** (Root, True).

Procedure **MAFIA_LO** (*C, IsHUT*) {

(1) *HUT* = *C*.head ∪ *C*.tail;

(2) IF Length(*HUT*) ≦ Length(*lfi*)

(3) THEN stop generation of children and return;

(4) Count all children, use PEP to trim the tail, and reorder by increasing support;

(5) *PEPSet* = {items moved from tail to head};

(6) *C*.head = *C*.head ∪ *PEPSet*;

(7) FOR EACH item *i* in *C*.trimmed_tail DO {

(8) *IsHUT* = whether *i* is the first item in the tail;

(9) *NewNode* = *C* ∪ {*i*};

(10) **MAFIA_LO** (*NewNode*, *IsHUT*) }; // end of for each

(11) IF *IsHUT* and all extensions are frequent

(12) THEN stop exploring this subtree and go back up tree to when *IsHUT* was changed to *True*;

(13) IF *C* is a leaf and Length(*C*.head) > Length(*lfi*)

(14) THEN *lfi* = *C*.head

} // end of procedure

Figure A.1: The MAFIA_LO Algorithm

The MAFIA_LO algorithm is presented in Figure A.1. Differences from the original MAFIA algorithm are highlighted by underlining. Line (2) modifies the original HUTMFI pruning (original statement is “IF HUT is in MFI”). If a node **c**’s HUT (head union tail) is discovered to be no longer than the longest frequent itemset found so far, we never have to explore any subset of the HUT, and thus we can prune the entire subtree rooted at node **c**. Line (13) and (14) find a longer frequent itemset and perform the updating (original statements are “IF (C is a leaf and C.head is not in MFI) THEN Add C.head to MFI”). For more information about the original algorithm, please refer to [BCG01].

- **The MAFIA_LO_ALL Algorithm**

Input: *C*: the current node, *IsHUT*: whether this is a HUT check

Global: *LFIList*: the set of longest frequent itemsets found so far

LFILen: the length of longest frequent itemsets found so far

Local: *HUT*: the set of head & tail, *NewNode*: a node

PEPSet: a set of items that are moved from tail to head by PEP pruning

Output: The *LFIList* that is set of all the longest frequent itemsets

Method: Call **MAFIA_LO_ALL** (Root, True).

Procedure **MAFIA_LO_ALL** (*C, IsHUT*) {

(1) *HUT* = *C*.head ∪ *C*.tail;

(2) IF Length(*HUT*) < *LFILen*

(3) THEN stop generation of children and return;

(4) Count all children, use PEP to trim the tail, and reorder by increasing support;

(5) *PEPSet* = {items moved from tail to head};

(6) *C*.head = *C*.head ∪ *PEPSet*;

(7) FOR EACH item *i* in *C*.trimmed_tail DO {

(8) *IsHUT* = whether *i* is the first item in the tail;

(9) *NewNode* = *C* ∪ {*i*};

(10) **MAFIA_LO_ALL** (*NewNode*, *IsHUT*) }; // end of for each

(11) IF *IsHUT* and all extensions are frequent

(12) THEN stop exploring this subtree and go back up tree to when *IsHUT* was changed to *True*;

(13) IF *C* is a leaf and Length(*C*.head) ≧ *LFILen*

(14) THEN IF Length(*C*.head) > *LFILen*

(15) THEN{ Empty *LFIList*;

(16) Insert *C*.head into *LFIList*;

(17) Update *LFILen* with Length(*C*.head); }

(18) ELSE insert *C*.head into *LFIList*; } // end of procedure

Figure A.2: The MAFIA_LO_ALL Algorithm

- **The FPMAX_LO_ALL Algorithm**

Input: *T*: an FP-tree
Global: *Head*: a list of items; *Tree*: the initial FP-tree
LFIList: the set of longest frequent itemsets found so far
LFILen: the length of longest frequent itemsets found so far
Output: The *LFIList* that is set of all the longest frequent itemsets
Method: Call **FPMAX_LO_ALL** (*Tree*).

```
Procedure FPMAX_LO_ALL (T) {
(1)  IF T only contains a single path P
(2)  THEN IF Length(Head ∪ P) > LFILen
(3)          THEN {
(4)                  Empty LFIList;
(5)                  Insert Head ∪ P into LFIList;
(6)                  Update LFILen with Length(Head ∪ P); }
(7)          ELSE insert Head ∪ P into LFIList;
(8)  ELSE FOR EACH item i in header table of T DO {
(9)          Append i to Head;
(10)         Construct Head's conditional pattern base;
(11)         Tail = {frequent items in base};
(12)         IF Length(Head ∪ Tail) ≧ LFILen
(13)         THEN {
(14)                 Construct Head's conditional FP-tree T_Head;
(15)                 FPMAX_LO_ALL (T_Head); }
(16)         Remove i from Head. } // end of for each
}    // end of procedure
```

Figure A.3: The FPMAX_LO_ALL Algorithm